MICROSOFT

EDITOR/ ASSEMBLER-PLUS

EDITOR/ ASSEMBLER-PLUS

Produced by Microsoft
Written by Mark L. Chamberlin and William E. Yates
Instruction Booklet by William Barden Jr.

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

COPYRIGHT NOTICE

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

Table of Contents

Chapter One

Getting Started With EDTASM-PLUS

Chapter Two

EDTASM-Plus Editor

Chapter Three

EDTASM-PLUS Assembler

Chapter Four

Assembler Macro Capability

Chapter Five

EDTASM-PLUS Z-BUG

Chapter One

Getting Started With EDTASM-PLUS

- What is Editor/Assembler-Plus?
- The Designers
- A Word About Microsoft
- The Right Hardware
- EDTASM-PLUS Cassette
- How To Load EDTASM-PLUS
- What To Do About Loading Problems
- EDTASM-PLUS Line Input and Output
- RESET Recovery Procedure

What is Editor/Assembler-Plus?

Editor/Assembler-Plus (EDTASM-PLUS) is a software package, supplied on cassette tape, that includes a complete Editor, Macro Assembler, and Debug program. The Editor and Assembler include Radio Shack's EDTASM commands as a subset, but add many powerful features to the basic capability of that package. The Debug package, called Z-BUG, is a completely new design of a "big-system" debugger.

EDTASM-PLUS is designed to be used on a TRS-80 Model I Computer with Level II BASIC and 16K or more of RAM memory.

EDTASM-PLUS gives you the capability to assemble TRS-80 assembly language programs directly **into** memory, without having to write out object programs to cassette tape and then load them. In addition, EDTASM-PLUS provides a **macro** capability to automatically generate in-line code by invoking an assembly-language macro, or set of predefined instructions.

As the Editor, Assembler, and Z-BUG are all **resident** at one time, it is a simple matter to efficiently edit source programs, assemble them into RAM memory, and then debug them. This process can be repeated over and over until the assembly-language program is checked out; the time required to switch between Editor and Assembler, between Assembler and Z-BUG, and from Z-BUG to Editor is a fraction of a second.

The Editor portion of EDTASM-PLUS includes the basic commands of EDTASM, but adds such features as line number offsets, new line range specifications, a block edit, moves and copies of blocks, string find and substitution commands, automatic line extension, and the ability to "quash" portions of EDTASM-PLUS to obtain additional text buffer area.

The Z-BUG debugger of EDTASM-PLUS provides functions normally only found in larger computer systems, including **symbolic debugging** based on the assembler symbol table, byte, word, mnemonic, and ASCII formats, up to eight breakpoints, user-defined number bases, and single instruction stepping.

The people at Microsoft hope you will be pleased with these new tools to facilitate your assembly-language programming of the TRS-80.

The Designers

Microsoft Editor/Assembler-Plus was written by Mark L. Chamberlin and William E. Yates, consultants to Microsoft. Mr. Chamberlin is the author of the original Radio Shack Editor/Assembler.

A Word About Microsoft

Microsoft produces high-quality, concise software for today's microprocessors.

Microsoft's BASIC Interpreter, in its several versions, has become the standard high-level programming language used in microcomputers. In addition to Radio Shack TRS-80 Level II BASIC, and TRS-80 Disk BASIC, Microsoft has supplied BASIC Interpreters for the Commodore PET, the Apple II Computer, NCR 7200, Compucolor II, OSI, Pertec Altair, and many others.

Microsoft's careful approach to the development of microprocessor software has allowed the production of large amounts of bug-free, well-designed code in a minimum amount of time. Currently available: BASIC interpreters for the 8080, 6800, and 6502 microprocessors; a FORTRAN compiler, assembler, loader and runtime library package for the 8080, and Z-80 microprocessors; an ANS-74 COBOL compiler for the 8080 and Z-80; and a complete offering of systems software for the new 16-bit microprocessors.

Microsoft Consumer Products was founded as a division of Microsoft in the summer of 1979 to provide microcomputer users with high quality system and utility software as well as application software.

EDTASM-PLUS is just one of many Microsoft products being planned for the end-user or consumer market. All of these software packages will be marketed by Microsoft Consumer Products.

Microsoft Consumer Products is dedicated to providing only the best, most reliable microcomputer software.

For more information on Microsoft Consumer Products software, please write to:

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

The Right Hardware

EDTASM-PLUS can be used with the Radio Shack TRS-80 Model I Microcomputer with Level II BASIC and 16K RAM minimum memory.

EDTASM-PLUS Cassette

The EDTASM-PLUS Cassette that comes in your EDTASM-PLUS package is a high quality recording from Microsoft.

The EDTASM-PLUS program is recorded two times on the side of the cassette tape with EDITOR/ASSEMBLER-PLUS printed on it. A length of leader tape precedes each recording. If you listen to the tape, you will hear a steady pilot tone during these leader sections.

Side Two of the cassette tape contains two recordings of Stand-Alone Z-BUG. Instructions for the use of Stand-Alone Z-BUG are given in Chapter 5.

How to Load EDTASM-PLUS

To load EDTASM-PLUS from cassette, use the following instructions:

1. Put the EDTASM-PLUS cassette into the TRS-80 recorder so that the side with Editor/Assembler-Plus is facing up.

2. Rewind the tape to its beginning.

3. Enter the command, SYSTEM, and press the [ENTER] key.

4. In response to the *? prompt, enter: EDTASM [ENTER].

5. Press the PLAY button on the TRS-80 recorder. There should soon be two asterisks in the upper-right corner of the TRS-80 screen (the one on the right "blinks"). These two asterisks signify that EDTASM-PLUS is loading.

6. A successful load will result in another *? prompt. In response to this prompt, enter: [/] [ENTER].

7. Assuming that all goes well, the screen will display a MICROSOFT COPYRIGHT notice, followed by an asterisk prompt character: *

WARNING:
Before loading EDTASM-PLUS or any other recordings into the TRS-80 microcomputer, we strongly urge you to disconnect the smallest gray plug that is normally inserted into the "MIC" jack of the tape recorder.

If for any reason during the actual reading of a tape the TRS-80 turns off the recorder (via the smallest gray plug), a "spike" may be recorded on the tape. Should this happen, the recording you are entering will be **permanently** damaged.

Our experience shows that this is most likely to occur when using the Radio Shack CTR-80 recorder, but we recommend that you still disconnect the smallest gray plug no matter what recorder you are using.

What To Do About Loading Problems

The TRS-80 is known to be "volume sensitive" when it comes to loading programs from cassette.

The most common loading problem is finding the correct settings for the TONE and VOLUME controls on the TRS-80 recorder. We suggest that you start with a low VOLUME setting and adjust it up one half level each time you attempt a load. The TONE control is not as important, but change it also.

Since the sensitivity of individual cassette recorders varies significantly, there is no way to determine specific settings. Once a tape loads, it is a good idea to write down the settings on the cassette label for future reference.

If you still cannot load a tape, you might try cleaning and demagnetizing the head of your TRS-80 recorder. Use a high-quality head cleaner and an inexpensive head demagnetizer for these tasks. Both can be purchased at Radio Shack or many other electronics outlets for under $10 (at the time of this writing). We don't recommend so-called "cleaner tapes" as they are often abrasive and may damage the head of your recorder.

Other suggestions to try before taking the matter up with your Radio Shack dealer include the following:

1. Try loading the second recording on your EDTASM-PLUS cassette.
2. Try loading the tape with a different cassette recorder.
3. Dust and other particles can sometimes prevent a load. To remove particles, run the tape through REWIND and FAST FORWARD a few times.
4. Don't try loading the TRS-80 when you first turn it on. Let it warm up a few minutes instead.
5. Remove the earphone jack and run the tape to listen for the leader tones and digital signals of the files. If you don't hear these sounds, try a different recorder. If you still don't hear them, chances are you have a blank tape.

6. Ask your Radio Shack dealer about Radio Shack's "cassette modification" fix. This hardware correction should make your EDTASM-PLUS less dependent upon exact VOLUME settings.

EDTASM-PLUS Line Input and Output

EDTASM-PLUS uses a **backspace** [←] to delete the previous character just as is done in other TRS-80 programs. The entire input line may also be deleted by means of a [SHIFT] left arrow [←] .

Portions of EDTASM-PLUS use an "ESCAPE" character, which is entered by a [SHIFT] up arrow [↑] , and displayed as a "$". The ESCAPE character cannot be deleted by a backspace or line deletion in Z-BUG.

The [SHIFT] [@] command will "hold" the display during rapid listing of lines. Hitting any key except [BREAK] will restart the display.

The [BREAK] key can generally be used to return to the EDTASM-PLUS "command" level at any point during the middle of line input or after the messages "READY PRINTER" or "READY CASSETTE". This is a means to effectively cancel the current line or action.

RESET Recovery Procedure

Under certain conditions control over EDTASM-PLUS may be lost. This is a (relatively) common occurrence during user program debugging when the user has not anticipated all of his program's actions. EDTASM-PLUS may also infrequently "hang" during line printer or cassette I/O due to hardware malfunctions.

To regain control at any time, press [BREAK] . If [BREAK] does not return control to ESTASM-PLUS, use the following procedure:

1. If using a system with no expansion interface, press [RESET] on the left rear of the cpu. If using a system **with** an expansion interface, press [RESET] while holding down [BREAK] .

2. Enter [ENTER] for "MEMORY SIZE?" if your system has an expansion interface.

3. Enter SYSTEM mode by inputting "SYSTEM" in response to the LEVEL II BASIC " > " prompt.

4. Enter one of the following addresses to reenter EDTASM-PLUS:

   a . /17280 restarts EDTASM-PLUS and destroys the contents of the edit buffer.

   b . /17283 restarts EDTASM-PLUS with the contents of the edit buffer preserved.

   c . /17286 restarts EDTASM-PLUS with Z-BUG breakpoints and the contents of the edit buffer preserved.

Bear in mind that if control was lost during debugging, portions of EDTASM-PLUS and/or the user program may have been destroyed and further use of EDTASM-PLUS without reloading may cause unpredictable results!

Chapter Two

EDTASM-PLUS Editor

- A Review of the Basic Editor Commands
- A Review of the Editor Subcommands
- A New Way To Specify Ranges of Lines
- Specifying Line Number Offsets
- A Way To Edit a Set of Lines Without Individual E Commands
- How To Move and Copy Blocks of Lines
- Finding Strings and Substituting a New String for an Old
- Automatic Extend (X) Subcommands Over a Range
- Using the Quash Command To Obtain More Memory for Edits
- Editor Error Messages

A Review of the Basic Editor Commands

The Editor portion of EDTASM-PLUS adds many new and powerful commands to the basic Editor commands. Let's review the basic Editor operation and commands, and then discuss the additional capabilities.

The Editor builds a buffer of text data in RAM memory. Usually this text data represents assembly-language source lines, although it can be any text data. Each source line is made up of four segments, or fields. The first field is the optional label field, the second is the Z-80 instruction operation code or pseudo-operation code, the third is the operands for the operation, and the fourth field is the optional remarks for the instruction. The typical assembly source line

```
START   LD      A,23    ;LOAD THE A REGISTER WITH 23
```

has a label of "START", an operation code of "LD", operands of "A" and "23", and remarks starting with ";LOAD". Note that the remarks field is always started by a semicolon (";").

The Editor commands are used to read or write a source file from cassette tape, to insert or delete source code lines in the edit buffer, to replace source code lines with new lines, to display lines in the text buffer on the video display or to print them on the system line printer, to find a given character string in the text buffer, and to position the cursor to the proper line and tab position along the line.

Let's construct a short source file and write it to cassette to show how the commands are used. Load EDTASM-PLUS as described in the first chapter. After a successful load, the title should be displayed, followed by the "*" prompt character.

A two-line source file can be created in the text buffer by using the I(nsert) command:

```
*I100,10
00100 START     LD      A,23    ;LOAD THE A REGISTER WITH 23
00110           END     START   ;START ADDRESS FOR LOADER
00120           (press BREAK )
```

The "100" in the I(nsert) command is the starting line number for the insert, while the "10" is the number of lines to increment for each new

line. Both the starting line number and increment are optional; the default starting line is 100 and the default increment is 10 on initialization, or the current line and last entered increment after use. Now the source file of two lines may be written to cassette by the W(rite) command:

```
*W TWOLN
READY CASSETTE          (hit any key except BREAK )
*
```

The source file from the edit buffer will be written out to cassette tape under the name "TWOLN". The name, by the way, is optional. If a name is not specified, no name will be used. If a name is used it may be 1 to 6 alphanumeric characters, starting with an alphabetic character.

The cassette file can now be read back into the edit buffer by the L(oad) command, after first deleting the two lines by the D(elete) command.

```
*D100:110               (delete the two lines)
*L TWOLN                (load the source file)
READY CASSETTE          (hit any key except BREAK )
*
```

The L command does not require a file name. If none is given, the next file on the cassette is loaded. Notice that a **range** of lines was specified by the colon (":") between the starting and ending line numbers for the delete.

To verify that the file has been loaded properly from cassette, we can display the edit buffer by the P(rint) command:

```
*P#:*                   (display the buffer)
```

The special symbols "#" and "*" represent the first line of the edit buffer and the last line of the edit buffer, respectively. We can use these symbols rather than having to remember the actual line numbers.

If our system has a line printer, the contents of the edit buffer may be printed by the H command

```
*H#:*                   (print buffer on line printer)
READY PRINTER           (hit any key except BREAK )
*
```

A variation on the H command allows only the text of the edit buffer to be printed—no line numbers are output:

```
*T#:*          (print text only)
```

To insert additional lines between two existing lines, another insert command may be performed. To insert the line

```
                LD      (3E20H),A     ;STORE IN CENTER OF SCREEN
```

between lines 100 and 110, for example, the I(nsert) below could be done

```
*I105
00105           LD      (3E20H),A     ;STORE IN CENTER OF SCREEN
NO ROOM BETWEEN LINES
*
```

The Editor assumed the increment value was 10 as it was not specified, accepted the first line, and then gave an error message when the next line, 115, would not fit in between lines 105 and 110.

There comes a time in every programmer's life when all the lines between lines 100 and 110 have been used. As the Editor does not use fractional line numbers, the source lines in the edit buffer must be renumbered by the N(umber) command:

```
*N100,10
*P#:*
00100 START     LD      A,23          ;LOAD THE A REGISTER WITH 23
00110           LD      (3E20H),A     ;STORE IN CENTER OF SCREEN
00120           END     START         ;START ADDRESS FOR LOADER
*
```

In this example, the renumber command was used to renumber the three lines with a starting line number of 100 and an increment of 10.

The N(umber) line command is also handy when two or more edit files are L(oaded) into the edit buffer. The Editor does not "interlace" the lines by line number, but appends each successive file. The N command will automatically renumber the entire edit buffer properly in such a case.

The R(eplace) command is used to replace one line with another line or group of lines. We can replace the first line by another, for example, in the sequence:


```
*R100,10
00100 START     LD      A,41H   ;LOAD A WITH ASCII 'A'
NO ROOM BETWEEN LINES
*
```

Here again, if the Editor "runs out of room" between lines, it will respond with the "NO ROOM..." message. We could have replaced line 100 with up to nine lines by:

```
*R100,1
00100 . . . .
00101 . . . .
    .
    .
```

Positioning commands enable us to find text within the edit buffer. The Editor maintains a pointer to the current edit line of the edit buffer. When the symbol "#" is specified, the Editor line pointer is set to the first line of the edit buffer. Specifying "*" sets the Editor line pointer to the last line of the edit buffer. We've seen how these symbols are used when we printed the entire edit buffer by the P#:* command.

A third symbol, the period, ".", refers to the current line in the edit buffer. We can scroll up and down the entire edit buffer by using the up [↑] and down [↓] arrows. Once positioned to the proper line, we can use the "." symbol to reference the current line as in the print command

```
*P.
```

which displays the current edit buffer line on the screen.

Another way of positioning the Editor line pointer is by F(inding) a known character string in the edit buffer. The character string we would typically search for would be a unique string. If a label for a line of our source code was called by a unique name such as "TST34", for example, we could perform the following command to find the line:

```
*F$TST34
00550 TST34     CP   50       ;TEST FOR A LESS THAN 50
*P.
00550 TST34     CP   50       ;TEST FOR A LESS THAN 50
*
```

The "$" in the command F$TST34 is not a dollar sign character! It is used to represent an "ESCAPE" character, which in the TRS-80 is equivalent to a [SHIFT], up arrow [↑] . It is used here as a "delimiter" (see "Finding Strings..." later in this chapter).

The line in which the character string appears is printed out with its number. Displaying the current line (".") by a P. would again print the line. If the string is not found, the message "STRING NOT FOUND" is displayed.

Notice that we must be aware of the position of the current line in the edit buffer. The search for the F command goes **forward**, so it is best to scroll back by using the up arrow [↑] or to perform a P# to position the line pointer before the expected position of the search string or at the very top of the edit buffer.

The search string in the F command may be any 1 to 16 character string that appears somewhere in the edit buffer. If the string appears more than one time in the edit buffer, the F(ind) command may still be used; the Editor will display the next occurrence of the string.

All of the above Edit commands allow us to add, replace, delete, display, print, renumber, or otherwise manipulate edit buffer **lines**. In the next section we'll review how a line may be edited on a **character by character** basis.

The Editor commands for line manipulation are shown below. These commands do **not** include the new EDTASM-PLUS commands and features that we will be discussing later in the chapter.

Command	Format	Sample	
Basic	B	*B	Return to Level II BASIC
Delete	D[line1 [:line2]]	*D100:*	Deletes line 100 through end of buffer
Find	F$[string]	*F$RALPH	Finds string "RALPH". ("$" is [SHIFT] [↑])
Insert	I[line1 [,inc]]	*I109,3	Insert following lines starting at line 109 and incrementing by 3 (112,115,...)

Command	Format	Sample	
Hardcopy	H[line1 [:line2]]	*H#:221	Output lines from start of buffer through 221 to printer
Load	L[␣filename]	*L TEST	Load file "TEST" from cassette tape
Number	N[line [,inc]]	*N100,5	Renumber lines starting at line 100 and incrementing by 5 (105,110,...)
Print	P	*P	Display the next 16 lines
Print	P[line1 [:line2]]	*P100:123	Display lines 100 through 123
Replace	R[line [,inc]]	*R300,3	Replace line 300 with following lines
Type	T[line1 [:line2]]	*T#:*	Print only text on system line printer
Write	W[␣filename]	*W JOHN	Write edit buffer to cassette tape with name "JOHN"
[↑] [↓]			Scroll up [↑] or down [↓]

Notes:

1. Brackets indicate optional arguments and ␣ indicates optional blanks.
2. "Line" defaults to current line if none entered.
3. "Inc" defaults to last entered increment if none entered.

A Review of the Editor Subcommands

The Editor subcommands are **commands** that allow characters within an edit buffer line to be edited on a character by character basis. Of course, a line could simply be edited by deletion and insertion using the Editor commands, but if only a portion of a line is to be edited, then the subcommands are much more efficient.

The subcommand mode is entered by the Editor command "E". To edit line number 552, for example, the command entry would be:

```
*E552
00552 _
```

Notice that after the command is entered, the line number of the line to be edited is displayed and the cursor positioned on the first character position of the line, ready for the first subcommand.

If you have Level II BASIC you will see that the Editor subcommands are identical to those in the Edit mode of Level II BASIC.

We will use a well known programming axiom to illustrate subcommand use:

```
00522 _
```

The underline marks the cursor position, and the shaded area denotes the portion of the line that is **not displayed, but in the buffer.**

We can **position the cursor** along the line by the [SPACE BAR] and by the left arrow [←]. The [SPACE BAR] moves the cursor to the right, and the left arrow moves the cursor to the left. If a number precedes the space or left arrow, then the cursor moves that number of spaces to the right or left. Entering 20 followed by [SPACE BAR], for example, displays

```
00522 PROGRAMS TEND TO OCC_
```

Entering 3 followed by a left arrow [←] then displays

```
00522 PROGRAMS TEND TO __
```

Once the cursor is positioned we can delete, modify, or insert characters in the same manner we performed these operations on lines.

To insert characters, the "I" subcommand is used. The cursor is positioned on the character of the line **before** the point at which new characters are to be inserted. Then the I command is entered, followed by the characters to be inserted. When all the characters desired have been entered, a [SHIFT] up arrow [↑] is entered. **The** [SHIFT] [↑] is used to tell the editor that we wish to escape from the edit mode subcommand. This special character is necessary because the insert accepts **all** normal characters as characters to be inserted.

The [SHIFT] [↑] may be used at any time to end the insert and other edit modes.

To see how the I subcommand works, let's add a word before "MORE" in our axiom. First position the cursor:

```
00522 PROGRAMS TEND TO OCCUPY_
```

Now, type I, followed by "bMUCH" followed by [SHIFT] [↑] :

```
00522 PROGRAMS TEND TO OCCUPY MUCH_
```

Deletions are performed by positioning the cursor on the character to be deleted and entering "D". If more than one character is to be deleted, a value can be entered before the D. To delete "MUCH", position the cursor on the M of MUCH, enter 5D, and the string will be deleted.

```
00522 PROGRAMS TEND TO OCCUPY _
```

The format for **C(hange)** is similar to that for deletions. If the C is preceded by a value, then the specified number of characters are to be replaced by a string that follows the "C". If no value is used, then only one character is specified for the change.

To change "MORE" to "LESS" in the sample axiom, enter 4CLESS after positioning the cursor under the "M" in "MORE". The result will be:

```
00522 PROGRAMS TEND TO OCCUPY LESS_
```

The **S(earch) subcommand** allows us to find a specified character in our test line. For example, we could search for the "Y" by entering "SY"

while we were in the Edit mode. If the cursor were positioned somewhere **before** the "Y", the S subcommand would result in:

```
00522 PROGRAMS TEND TO OCCUP_
```

If a value was used before the S, the **nth occurrence** of the specified character would be found. If we were at the beginning of the line and "3SE" was entered, for example, the result would be:

```
00522 PROGRAMS TEND TO OCCUPY MORE M_
```

Two other **positioning commands** are L and X. The L subcommand is used to print the entire line and then position the cursor at the beginning of the line. This is useful to see the remainder of the line at any time:

```
00522 PROGRAMS TEND TO OCCUP_

00522 PROGRAMS TEND TO OCCUPY MORE MEMORY THAN IS
AVAILABLE.
00522 _
```

The X subcommand does somewhat the same thing, in that it displays the entire line, but it then positions the cursor to the end of the line and sets the insert mode. This is useful for adding to the end of the line:

```
00522 PROGRAMS TEND TO OCCUPY MORE MEMORY THAN IS
AVAILABLE._
```

Two somewhat violent commands, H(ack) and K(ill), allow the user to delete a segment of a line. The H subcommand hacks off the remainder of the line from the current cursor position and sets the insert mode. Hacking here:

```
00522 PROGRAMS TEND TO OCCUP_
```

causes the following result, with the insert mode active:

```
00522 PROGRAMS TEND TO OCCUP_
```

K deletes a segment of a line up to the nth occurrence of a specified character. The nth occurrence of the character is found as in the S subcommand. If the cursor is at the start of the line and "2KM" is entered, the following line results:


```
00522 _
```

There are a number of subcommands to **restart or terminate** the Edit mode. If a mistake is made while editing a line and the user wishes to cancel all previous editing changes, the **A**(gain) subcommand can be given to restart the edit with the cursor positioned at the beginning of the line as in:

```
00522 PROGRAMS TEND TO OCCUPY MORE MEMORY THAN
!_
00522 _
```

If the line was mistakenly edited and the user wishes to simply restore the old line, then a **Q(uit)** can be entered to ignore changes.

If all of the changes have been made correctly, then an **E(nd edit)** or simply the [ENTER] key can be used to enter the editing changes.

```
00522 PROGRAMS TEND TO OCCUPY MORE MEMORY AND TIME
THAN AVAILABLE.
*
```

The subcommands for the Edit mode are shown below.

Subcommand	Format	Description
Again	A	Cancel all changes and restart
Backspace	n←	Move cursor n positions left
Change	nCstring	Change n characters at current cursor position to string characters
Delete	nD	Delete n characters at current cursor position
End edit	E	End edit and enter all changes without displaying remainder of line
End edit	[ENTER]	End edit and enter all changes and display remainder of line
Extend line	X	Move to the end of the line and enter insert mode

Subcommand	Format	Description
Hack	H	Delete line from current cursor position to end and enter insert mode
Insert	Istring	Enter insert mode before current cursor position. Terminate with SHIFT , up arrow ↑ .
Kill	nKs	Kill all characters from current cursor position to nth occurrence of "s"
List line	L	List line and position cursor to start of line in following line
Quit	Q	Quit Edit mode and ignore all changes
Search	nSs	Search for the nth occurrence of "s"
Space	nspace	Move cursor n positions right.

A New Way To Specify Ranges of Lines

EDTASM-PLUS extends the way that line number ranges can be specified. In the previous version of EDTASM, a colon was used when a range of lines was specified, as in *P100:300 which displayed all lines from 100 through 300. This method of specifying lines is still valid, but is supplemented by the ability to express a range by a starting number and the total number of lines in the range.

Using this specification, it is easy to get "the first five lines of the edit buffer" or the "next six lines following line 1001". This is especially handy when all of the line numbers are not known, as in a recently edited source program.

The format for this method of specification is

SLN!n

SLN is the "starting line number" and may be any valid line number for a line in the edit buffer. "!" is a symbol that designates that a line count rather than an ending line number will be used. "n" is the line count — the number of lines, including the starting line number, that are to be used in the range.

To display the first five lines of the edit buffer, for instance, we can enter

*P#!5

To delete the 10 lines in the edit buffer starting with line number 1011, we enter

*D1011!10

Of course, just as in the case of the "starting line:ending line" specification, the range of lines must be **in** the edit buffer or a "BAD LINE NUMBER" error will result. If the range of lines is partially in the edit buffer, or if the starting line number does not exist, then the Editor works on all the lines within the range it can find, as one would expect.

The "for how many lines" specification may be used in place of the "starting line number:ending line number" specification for any Edit command.

Specifying Line Number Offsets

EDTASM-PLUS provides a way to reference lines by offsets from a given line number. This feature makes it possible to reference the "line five lines above line 1770" or the "line 15 lines after line 2022". Here again, as in the case of the "how many lines" specification, lines can be referenced without knowing all the line numbers in a set of lines.

This type of line reference is very handy in **positioning the Edit line pointer** prior to searches, displays or prints. For example, if we want to start a search 5 lines back from the current line we could enter the following sequence:

```
*P.-5
00550 THIS IS LINE 550 DISPLAYED BY THE P.-5 COMMAND.
*F$LOST CHORD
00580 THIS IS THE LOST CHORD LINE.
*
```

The first command displays the current line less five lines and positions the Editor line pointer in preparation for the search. The F command searches for the LOST CHORD and displays the line when it is found.

The offset method of specifying line numbers can be used any time that a line number is called for. It can not only be used with the symbol for current line ("."), but also with the symbol for start of buffer ("#") and the symbol for end of buffer ("*"). To print the last **six** lines in the edit buffer, for instance, we could enter

```
*P*-5:*
```

and to print the first six lines at the beginning of the buffer we could enter

```
*P#:#+5
```

Another format, of course, is to use the offset with an actual line number as in

```
*P1010+5
```

Of course if the resulting line number is nonexistent, as in the expression #−2 or *+3, a "BAD LINE NUMBER" error message results.

A Way To Edit a Set of Lines Without Individual E Commands

EDTASM-PLUS adds a new way of using the E command. If there is a **set** of lines that requires editing, then a **collective E(dit)** command can be given to remain in the Edit mode for the entire range rather than entering an E for each line.

This is a feature that makes editing go much more smoothly when more than one line is to be edited. Suppose that lines 1010 through 1050 need to be edited. Using the collective edit command we simply say

E1010:1050 or E1010:1010+4 or E1010!5

The Editor will now stay in the Edit mode until lines 1010 through 1050 have been edited and will then return to the command driver.

If you've made a mistake and really don't want to remain through the entire range, simply type a **Q** to quit the Edit mode. The ENTER and **E** subcommands are used as before to enter the line after editing.

The new format for E, then, is used with any range specification when a set of **contiguous** lines is to be edited, and is not used when just one line is to be edited as before.

How To Move and Copy Blocks of Lines

Two new commands have been added to EDTASM-PLUS to enable the user to **move a block** (or range) of lines or to **copy a block** of lines.

The move command allows us to take a specified block of lines and move them to a new area of the edit buffer. The old block of lines is deleted. Previously, this action would have been accomplished by inserting the lines in the new area and then deleting the old lines. The **M**(ove) command is a powerful feature as it allows us to rearrange sections of assembly-language programs easily.

The format of the move command is

MTLN,RNG,INC (increment optional)

"TLN" is the "target" line number, the starting line number for the new block. "RNG" is the specification for the old block of lines; RNG can use any of the range specifications that we have mentioned previously, as, for example, 100:200, 100:100+5, or 100!5. "INC" is the increment for line numbering on the new block.

As an example, suppose that we want to make a subroutine out of some assembly language code to shift the HL register pair four bit positions left. The initial code is:

```
*P500:500+3
00500 SHIFT   ADD   HL,HL      ;SHIFT HL 1 BIT LEFT
00510         ADD   HL,HL      ;SHIFT HL 2 BITS LEFT
00520         ADD   HL,HL      ;SHIFT HL 3 BITS LEFT
00530         ADD   HL,HL      ;SHIFT HL 4 BITS LEFT
*
```

This code is "imbedded" in other code, and we would like to move it to a common subroutine area at the end of all other code. We can easily move the code to a new area starting with line 10000 by:

```
*M10000,500!4,10
```

This move transfers lines 500 through 530 (500!4) into a new block of lines 10000 through 10030. The target line number was 10000, and the increment for the new block was 10. The old lines 500 through 530 were deleted after the move.

About the only thing to watch for in the move is that the new range of lines does not conflict with existing lines in the edit buffer. If there is a previous line number which is the same number as the starting line number in the move command, the Editor will attempt to make the move by starting at the starting line number plus the increment. If line number 10000 already was used in the previous example, the new block would start at 10010.

If there is not enough room to add all of the lines in the move, a "NO ROOM BETWEEN LINES" error will result. This would have happened in the previous example if line 10030 had been in use, for instance.

The copy command is similar in format and concept to the move. The only difference between the move and **C**(opy) commands is that move deletes the old block of lines, but copy keeps them intact. The format of the copy is:


```
CTLN,RNG,INC                (increment optional)
```

We could have copied the lines of the previous example by:

```
*C10000,500!4,1
```

This command would have moved the four lines at 500 through 530 into the new block of 10000 through 10003, but retained the lines at 500 through 530, assuming a previous (default) increment of 10.

Just as in the move command, if there is not enough room to insert all of the lines in the new block, a "NOT ENOUGH ROOM" error results. Likewise, the starting line number is changed to the target line plus the increment if the target line number already exists.

"INC" defaults to the current increment if not specified, for both Move and Copy.

Finding Strings and Substituting a New String for an Old

We have already discussed the F(ind) command in our review of the basic Editor commands. **EDTASM-PLUS extends the usefulness** of this command with its new methods of specifying ranges of lines. Recall that in the previous approach, the Editor line pointer must be positioned to a line **before** the block of lines to be searched. In EDTASM-PLUS, a **range** of lines can be specified for the search.

The format for this type of search is:

```
FRNG$string
```

"RNG" is a range specification in any of the formats we have been using — 100:300, 100!15, or 100:100+23, for example. **(The string to be searched for is preceded by a "$" delimiter, which is not a dollar sign, but an ESCAPE character,** [SHIFT] [↑] **!)**

The benefit of this approach is that we do not have to first position the Editor line pointer; the search simply takes place over the specified range of lines.

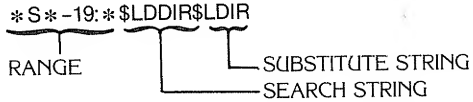
When a range is specified for a search, **all lines containing the search string are displayed.**

Suppose that we have mistakenly used the mnemonic "LDDIR" instead of the correct assembler mnemonic "LDIR". We can find all occurrences of the error in the last 20 lines of the edit buffer as follows:

```
F*-19:*$LDDIR
02005 MOVE1    LDDIR      ;MOVE THE BLOCK
02200          LDDIR      ;MOVE THE BUFFER
02300          LDDIR      ;MOVE HEAVEN AND EARTH
*
```

In the above code there were three lines that contained the character string "LDDIR". The three lines were displayed, and the Editor line pointer set to the last line to be displayed.

The Substitute command works pretty much the same as the F(ind) command when a range is specified. If we had wanted to substitute a string of "LDIR" for the erroneous string "LDDIR" we could have used the same format to specify the search string, but appended a string to be substituted:



The Editor would then have searched for the LDDIR string over the entire range and substituted the LDIR string at each occurrence of LDDIR. In this case the lines containing the newly substituted strings are displayed:

```
*S*-19:*$LDDIR$LDIR
02005 MOVE1    LDIR       ;MOVE THE BLOCK
02200          LDIR       ;MOVE THE BUFFER
02300          LDIR       ;MOVE HEAVEN AND EARTH
```

In both the F(ind) and S(ubstitute) commands, the specified strings for the search and replacement strings must be **16 characters or less**. If a substitute is performed and a new line results that is greater than 128 characters, a "NEW LINE TOO LONG" message is displayed. This can happen, of course, if the original line is long to begin with, and the substitute string is longer than the search string.

The S(ubstitute) command can be used without a range, just as the F(ind) command can be used without a range of lines. In this case,

the Editor finds the first occurrence of the string, makes the substitution of the second string, and then displays the line. The line pointer is set to the line containing the string. The format in this case is:

*S$string (where $ is the ESCAPE, SHIFT ↑)

The Find and Substitute commands may be used with a **new** range, but without a new string. In this case, the Editor simply uses the last entered string as a default string.

The F and S commands may be used **without a string(s) or range**. In this case the Editor searches for the current string from the current line.

The F and S commands are extremely powerful when used to find and correct erroneous mnemonics or operands in source code, an operation which is necessary far more frequently than assembly-language programmers would hope for!

Automatic Extend (X) Subcommands Over a Range

EDTASM-PLUS adds the capability of entering the extend (X) subcommand mode for a range of lines. One use for this feature is to add comments to lines of assembly-language source code.

The format of the Extend command is

XRNG (RNG **usually** specified but optional)

where RNG is a range of lines, specified by any of the methods that we have described above, for example 100:1000, 333-45:300, or 1000!12. If RNG is not specified, only the current line will be extended. After the range of lines has been entered, the Editor automatically enters the Edit mode for the first line, positions the cursor after the last character of the line, and sets the insert mode. Characters can then be added to the line (or other editing subcommands can be performed). When all characters have been added, an E or [ENTER] is input, and the Editor displays the next line of the range in the same fashion. This process is repeated over the entire range of lines that has been specified.

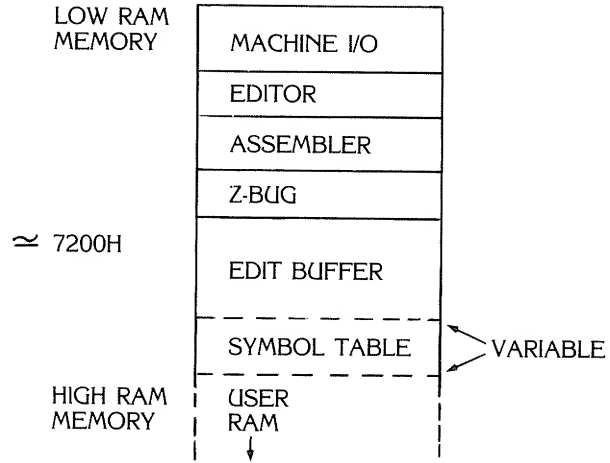
Let us see how this command works. Suppose that we wish to add comments to lines 1000 through 1020. The sequence used follows:

```
*X1000:1020
01000          LD      A,15_
      (comment added to above line, followed by [ENTER])
01010          LD      HL,BUFFER_
      (comment added to above line, followed by [ENTER])
01020          LD      B,40H
      (comment added to above line, followed by [ENTER])
*
```

To get out of the automatic Extend mode at any time, simply enter a Q in place of the E or [ENTER] after changes have been made. Remember that a Q or E must be preceded by a [SHIFT] [↑].

Using the Quash Command to Obtain More Memory for Edits

During normal operation in EDTASM-PLUS, the Editor, Assembler, and Z-BUG are "resident" at the same time. These three segments of EDTASM-PLUS use 12.5K bytes of RAM memory and are arranged as shown in the figure below:



The edit buffer uses available RAM memory for text storage. If more space is required to hold text in the edit buffer, either the Z-BUG or Assembler **and** Z-BUG areas may be released to the system for an extension to the edit buffer area.

The command for this action is the Quash command. If only the Z-BUG portion of the system is to be used for the edit buffer, then the

QZ

command is entered. If **both** the Z-BUG and Assembler portions are to be used as edit buffer area, then

QA

is entered.

Quashing Z-BUG adds an additional 3.2K bytes for the edit buffer, while quashing both Z-BUG and the Assembler adds an additional 8.1K bytes for editing.

The Quash command preserves the current text in the edit buffer so that it does not have to be reentered after the Quash.

Because this is a catastrophic action if the command is erroneously entered, EDTASM-PLUS asks the user "QUASH?" to make certain that he wants to quash. **If the user has made a mistake, a** BREAK **may be typed to abort the quash; hitting any other key causes the quash to take place.**

Of course once the Quash command has been acted upon, any Assembler or Z-BUG command will result in the error message "BAD COMMAND" if that function has indeed been quashed. QZ disables the Z command and QA disables the Z, A, Q, and O commands.

The Quash command is meant to provide more space for general text editing or for situations where there are too many source lines to be held without quashing a portion of the package. Bear in mind that if the assembler is also quashed, it probably will not be possible to assemble a source program even by reloading as there will not be enough space in the edit buffer to accomplish the reload. The most practical approach with the source programs is to quash only the Z-BUG portion, assemble the program, output the object to cassette tape, and then reload the object program with Stand-Alone Z-BUG for debugging.

Use the quash capability only when absolutely necessary, as its use deletes many of the powerful assembly and debug features of EDTASM-PLUS that we will be discussing shortly.

Editor Error Messages

The following is a list of error messages that may occur during Editor operation:

Message	Description and Corrective Action
BAD COMMAND	Editor does not recognize command. Is it valid?
BAD LINE NUMBER	Editor cannot find line number. Check to see if line is actually present in the edit buffer.
BAD PARAMETERS	Editor cannot decode the operands for the command. Check format.
BUFFER EMPTY	User has specified operation on a line when buffer is empty. Reload or reenter text.
BUFFER FULL	No more room in the edit buffer. Quash Z-BUG or Z-BUG and Assembler to get more space, or break up program into separate modules.
NEW LINE TOO LONG	Substitute has been performed that resulted in a line greater than 128 characters. Shorten the line.
NO ROOM BETWEEN LINES	Insert, move, or copy action resulted in too many lines to fit between existing lines. Renumber existing lines with larger increment.
STRING NOT FOUND	Find or substitute command specified a string that cannot be found. Check the search string for proper characters.

Chapter Three

EDTASM-PLUS Assembler

- A Review of the Assembler Operation
- A Review of Pseudo-Operations
- Assembler Switches
- Assembling Into Memory–Automatic Origin
- Assembling Into Memory–Manual Origin
- Assembling With an Absolute Origin
- Assembler Expression Evaluation
- Conditional Assembly

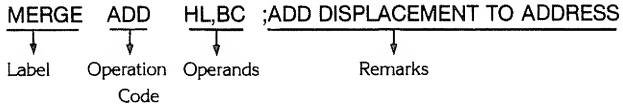
A Review of the Assembler Operation

EDTASM-PLUS adds many new features to the basic assembling ability of the Editor/Assembler. Before we discuss the new capabilities, let us review the basic assembly process, pseudo-operations, and assembly commands.

The Assembler operates from source code stored in the edit buffer. The source code is generated from the keyboard, using the Editor, or is loaded from a previously written source file on cassette tape.

The source code consists of **source lines.** Each line of source code generally produces one machine language instruction, although under certain conditions no instructions are generated, data is generated, or some other Assembler action takes place.

The usual assembly-language source line consists of four **fields:** an optional label, an operation code, operands, and optional remarks:



The operation code is the heart of the assembly-language line. It is a mnemonic name that represents one of the hundreds of individual instructions that are in the instruction repertoire (or set) of the Z-80 microprocessor used in the TRS-80.

Each of the instructions found in the Z-80 has a predefined format, and a **certain number of operands** associated with its operation. Some instructions require no operands, while others require two or three. Operands may be register names, data values in decimal or hexadecimal, symbols, expressions, or combinations of these things.

The remarks column is strictly optional. When used, the remarks field must start with a semi-colon (;) to delimit the remarks from other fields. The source line may also consist entirely of a remark with a starting semi-colon.

The label column is another optional column. Labels are used in lieu of BASIC line numbers so that one instruction may reference another, primarily for conditional and unconditional jumps.

The fields of each source line are **"free-form"** fields, that is, they do not have to start at specific columns. Each field of the source line need only be separated by a blank, and the remarks field must have a semi-colon delimiter. For convenience and readability, however, the right arrow will tab to predefined tab positions, and the four fields can be put into nice, neat columns during Edit.

Let us assemble a sample program to review how the Assembler operates. The source program below clears the display by storing blanks in all of the 1024 character positions. We will assume that the Editor has been used to produce this set of lines in the edit buffer.

```
00100          ORG   7000H
00110 CLEAR    LD    HL,3C00H   ;ADDRESS OF SCREEN START
00120          LD    BC,1024    ;COUNT OF POSITIONS
00130 LOOP     LD    A,20H      ;LOAD BLANK
00140          LD    (HL),A     ;STORE BLANK
00150          INC   HL         ;BUMP POINTER
00160          DEC   BC         ;DECREMENT COUNT
00170          LD    A,B        ;TEST DONE
00180          OR    C          ;MERGE LS BYTE
00190          JR    NZ,LOOP    ;GO IF MORE TO STORE
00200          END
```

The source code above represents the clear program in **symbolic form.** It cannot, of course, be loaded into the TRS-80 in this form for execution, but must first be translated from the symbolic source form above into machine language instructions that the Z-80 will recognize.

Two labels have been used — "LOOP" and "CLEAR". Many times we do not know where each instruction will reside in memory and for that reason do not assign absolute memory addresses to the instructions. The symbolic location of the LD A,20H instruction is at LOOP and we have referenced that location in the "Jump Relative if Not Zero" instruction, which will cause a jump to location LOOP 1023 times through the loop.

One important point about the symbolic form of the program: We could, in fact, sit down and laboriously hand assemble these instructions into machine language form by reference to the Z-80 manual. The symbolic form, however, relieves us of this chore and lets the Assembler do all of the calculations.

Entering the command "A" takes the contents of the edit buffer and translates the symbolic form of the program into machine code.

```
7000            00100           ORG   7000H
7000 21003C     00100 CLEAR     LD    HL,3C00H  ;ADDRESS OF SCREEN
                                                    START
7003 010004     00120           LD    BC,1024   ;COUNT OF POSITIONS
7006 3E20       00130 LOOP      LD    A,20H     ;LOAD BLANK
7008 77         00140           LD    (HL),A    ;STORE BLANK
7009 23         00150           INC   HL        ;BUMP POINTER
700A 0B         00160           DEC   BC        ;DECREMENT COUNT
700B 78         00170           LD    A,B       ;TEST DONE
700C B1         00180           OR    C         ;MERGE LS BYTE
700D 20F7       00190           JR    NZ,LOOP   ;GO IF MORE TO
                                                    STORE
0000            00200           END
00000   TOTAL ERRORS

CLEAR           7000
LOOP            7006
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

The assembly listing shown above consists of four parts. The right section is the "source line image," simply a repetition of the source code from the edit buffer. The third column from the left lists the corresponding source line number for each line of code.

The two columns on the left are the locations at which the instructions will reside, and the actual machine language form of the instructions. By specifying the special assembler "pseudo-operation" ORG, for origin, we have informed the assembler to assemble the code starting at location 7000H, 1024 bytes down from the top of RAM in a 16K system.

The machine language form of the instruction consists of one to four bytes of data. Each byte of data is represented by two hexadecimal

digits, so there may be from two to eight hex digits in the second column for the corresponding instruction. The location column on the far left is incremented to reflect the number of bytes in the instruction. A two-byte instruction, for example, adds two to the address for the next instruction.

The machine language code represented by the data in the second column is the **object** code produced by the assembler. This object code may be saved on cassette tape for reloading by a debug package or by the monitor (SYSTEM) mode logic in Level II or Disk BASIC. Alternatively, it may simply be kept in memory for on the spot debugging with Z-BUG. This **in-memory assembly** is not generally possible with assembler packages and is one of the powerful features of EDTASM-PLUS we will be talking about shortly.

A Review of Pseudo-Operations

When is an opcode not an opcode? When it is a pseudo-operation! As we mentioned earlier, most source lines cause one machine-language instruction to be generated. The exception to this, however, is source lines that do not contain instruction opcode mnemonics, but contain operations to be performed by the assembler. These instructions are called **pseudo-ops** as they are written in the opcode column of the source line.

We have seen two of the pseudo-ops in the sample program of this section, the **END** and **ORG** pseudo-ops.

The END pseudo-op simply marks the end of the assembly source code. The Assembler makes several **passes** to resolve **forward references** and the like, and it must know where the end of the source code is.

The **END** may have an argument that specifies the **starting address** for the program. This is used by both Level II BASIC SYSTEM mode and by the EDTASM-PLUS Z-BUG Load command as the default starting address.

The ORG pseudo-op tells the Assembler where the program will be loaded for execution. It is generally the first command in the

source code. The ORG may also be used within the source code to indicate where sections of the program are to reside, or for reserving blocks of storage (see discussion of DEFS).

The ORG is a necessity for the old version of EDTASM. In EDTASM-PLUS, however, although we may use ORG to assemble **absolute** programs, it is generally not necessary. (We will explain in detail later in this chapter.)

There are four pseudo-ops that are used to generate **data** in place of machine-language instructions. Data values generated by these pseudo-ops are used as general constants, table values, or other types of data.

The first of the four generates one byte of data at the current assembly location. The **DEF**ine **B**yte below will create a single byte of 100.

```
01000 HUND     DEFB  64H        ;GENERATE CONSTANT OF 100
```

A second pseudo-op, **DEF**ine **W**ord, generates **two bytes** (16 bits) of data. This data is **almost always address data, arranged least significant byte followed by most significant byte,** in standard Z-80 address format.

```
01001 LOCNT    DEFW  START      ;ADDRESS OF START
01002          DEFW  LAST       ;ADDRESS OF LAST
01003          DEFW  MID        ;ADDRESS OF IN BETWEEN
```

The third pseudo-op, **DEFB with a character operand in quotes,** is much like the normal DEFB except that an ASCII character is generated. ASCII is the seven bit code (most significant bit is a zero) used by all TRS-80 input and output devices for character representation.

```
02010          DEFB  'S'        ;GENERATE AN ASCII S
```

A related pseudo-op, **DEF**ine **M**essage, generates a string of ASCII characters used as a **message** for output to the screen or printer.

```
03020 MESSG1   DEFM  'I NEVER USED A PROGRAM I DIDNT LIKE'
```

All of the pseudo-ops in the last group **generated data** which would be loaded along with machine language instructions. The **DEFS** pseudo-op, however, does not generate data, but simply tells the

assembler to **reserve a block of storage locations.** The **DEF**ine **S**torage pseudo-op simply causes the Assembler location counter to be incremented by the number of bytes specified by the operand. When the object code is loaded, the loader bypasses the block defined in the DEFS, **storing nothing there.** This code shows the use of DEFS:

```
7000 C3CB70    03000          JP    NEXT   ;CONTINUE
7003           03010 TABLE    DEFS  200    ;RESERVE TABLE OF 200
                                             BYTES
70CB 3E02      03020 NEXT     LD    A,2    ;TWO TIMES THROUGH
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

Note that in the object code above, the LD instruction was assigned a location CBH, or 200 decimal bytes past the first instruction at 7000H through 7002H.

Another way of performing exactly the same action would be to use a second ORG pseudo-op to reset the assembler (and loader) location counter:

```
7000 C3CB70    00110          JP    NEXT   ;CONTINUE
7003           00120 TABLE    EQU   $      ;START OF TABLE
70CB           00130          ORG   $+200  ;RESERVE TABLE OF 200
                                             BYTES
70CB 3E02      00140 NEXT     LD    A,2    ;TWO TIMES THROUGH
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

In the above code we used another pseudo-op, the EQUate pseudo-op. This pseudo-op equated a label to the value of the current assembler location counter ("$"). As the assembler scans the source lines, it builds a table of symbols and associated values called the **symbol table.**

Many of the values in the symbol table are the locations of the names of instructions which are then filled in for jumps and other instructions as the numeric values. Other values equated with symbols, however, may represent different data. If we used the ASCII character for "turn on cursor" (14) many times in a program, we might choose to call it "ONCUR" by an EQU:

```
02000 ONCUR    EQU    14      ;TURN ON CURSOR CHARACTER
```

Thereafter, every time we wanted to load, compare, or utilize the character, we could use "ONCUR" in place of 14 as in:

```
04000          CP   ONCUR     ;TEST FOR 14 ON CURSOR CHAR
```

When the operand of an EQUate is "$", then the label used in the EQU is equated to the current Assembler location counter value, which is simply the address for which the next instruction or data will be assembled.

The last pseudo-op is **DEFL**. DEFL is used similarly to EQU, but the label used with DEFL may be **redefined** later in the source code. We could, for example, have:

```
02000 COUNT    DEFL   100     ;SET COUNT
                 |
                 |
(other source code)
                 |
                 |
04000 COUNT    DEFL   200     ;SET COUNT
```

Most of the pseudo-ops can use a general expression as an operand.

Assembler Switches

The basic command to start assembly is "A". When this command is given, the Assembler will rapidly assemble the code in the source buffer and display it on the screen as it does so. At the end of the listing on the screen, the **symbol table** for the assembly will be listed in alphabetical order.

The symbol table contains all user labels from the assembly. Each symbol is displayed along with the corresponding value for that symbol. In many cases this value represents the location of the symbol in the assembled program; in other cases the value is a result of an EQUate.

At the end of the listing of the assembly and symbol table, the Assembler is ready to output the **object** to cassette tape. The assembler asks "READY CASSETTE" so that the user can prepare the cassette. Hitting any key other than [BREAK] or [SHIFT] [@] after this prompt results in the object code of the assembly being output to cassette tape.

If no file name is used, the object file will be output with file name **"NONAME"**. If a name is used, the Assembler will use the name from the A command:

```
*A␣NAME                    (␣ is optional space)
```

There are various options, called "switches", that may be used with the assembler "A" command.

The output of object code can be disabled by the "/NO", **No Object** switch. Many times it is convenient to assemble on a trial basis before producing an object tape. Any assembly errors can then be corrected, and the object tape produced after all assembly errors have been dealt with. The format for this option is

```
*A/NO
```

The listing and/or symbol table output may be suppressed by the "NL" and "NS" switches. If a hardcopy of the listing is required, the "LP" switch causes the listing and symbol table output to be sent to the system line printer. An assembly with no symbol table printout, no object, and output to the line printer, for example, would be specified by:

* A/NO/LP/NS

The switches may be used in any order.

The "WE" switch causes the assembler to wait on errors. This switch is useful when the listing is being displayed on the screen, as the assembly is displayed so rapidly that the individual errors cannot be seen. Pressing any key except [BREAK] or [C] after the error wait causes the Assembler to continue and wait on errors. Pressing [BREAK] aborts the assembly. Pressing [C] continues the assembly without waiting on further errors.

The above switches were used in the previous version of EDTASM. They are still valid in EDTASM-PLUS. EDTASM-PLUS adds additional assembly switches that are used in the same fashion. The new switches are "IM" to assemble directly into memory, "MO" for manual origin, and "AO" for absolute origin. We will discuss these in the next section of this chapter, as they all relate to assembling programs directly into memory.

For reference, the Assembler switches are:

Switch	Description
/NO	Suppress object output
/NS	Suppress symbol table printout or display
/NL	Suppress listing printout or display
/LP	Output listing and symbol table to line printer
/WE	Wait on errors until key depressed
/IM	Assemble object code directly into memory
/MO	Assemble with user-specified origin
/AO	Assemble with absolute origin

Assembling Into Memory–Automatic Origin

EDTASM-PLUS adds a powerful new feature to the basic Editor/Assembler, the **ability to assemble source programs directly into memory.** This "assemble into memory" capability lets the user avoid the time consuming portion of assemblies during debugging — creating and loading an object tape. Furthermore, as the Z-BUG debugger is a part of EDTASM-PLUS, programs assembled into memory may be immediately debugged.

A second important quality of assembling into memory is that the assembler will automatically **relocate** the object code to a convenient portion of memory; the user is relieved of the task of figuring out where he should place the object code.

Typically, the assemble into memory feature can be used during the debugging phase of assembly-language program development. Once the program has been fully debugged, it can then be output in final form to cassette tape for subsequent loads by the SYSTEM command in BASIC.

To see how this feature works, let us assemble a sample program. The command A/IM assembles the program from page 43 and produces the listing shown below:

```
71AD 21003C    00110 CLEAR  LD   HL,3C00H  ;ADDRESS OF
                                              SCREEN START
71B0 010004    00120        LD   BC,1024   ;COUNT OF POSI-
                                              TIONS
71B3 3E20      00130 LOOP   LD   A,20H     ;LOAD BLANK
71B5 77        00140        LD   (HL),A    ;STORE BLANK
71B6 23        00150        INC  HL        ;BUMP POINTER
71B7 0B        00160        DEC  BC        ;DECREMENT
                                              COUNT
71B8 78        00170        LD   A,B       ;TEST DONE
71B9 B1        00180        OR   C         ;MERGE LS BYTE
71BA 20F7      00190        JR   NZ,LOOP   ;GO IF MORE TO
                                              STORE
0000           00200        END
00000 TOTAL ERRORS
CLEAR  71AD
LOOP   71B3
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

One of the statements that we left out of the above assembly was the ORG, which specified the origin of the program. This was left out on purpose, as the Assembler automatically determines the origin for in memory assembly; it uses the first available memory location above the space used for EDTASM-PLUS, the Edit buffer, and the symbol table.

We can see from the listing that this program was assembled starting at location 71ADH. (This location may vary depending upon the version of EDTASM-PLUS that you are using.)

If we look back to the figure in Chapter 2, we can see how the assembler allocates memory for in memory assembly. The areas for machine I/O, Editor, Assembler, and Z-BUG are fixed in size. The Edit buffer, of course, holds whatever source program we have entered and is therefore variable size. The symbol table is also variable size and depends on the number of symbols that we are using in our source program. Each symbol takes from 4 to 9 bytes, dependent upon the number of characters in the symbol.

As the Assembler generates object code, it is built upward towards high memory from the symbol table end. The total area available for long assemblies ranges from the automatic origin to the end of RAM memory in the TRS-80. In a minimum system, this would be up to location 7FFFH; in a maximum system the upper limit would be FFFFH.

ORG pseudo-ops should, in general, **not be used** when assembling into memory with an automatic origin. If they **are** used, the effective address of the ORG will be the ORG address **plus** the value of the automatic origin. If the automatic origin is 71ADH, for example, the pseudo-op

```
ORG   200H
```

produces an origin of 73ADH, which is probably meaningless unless the user plays special tricks to reference data or code to the start of the program.

Assembling Into Memory–Manual Origin

It is possible for the user to specify a **manual origin** in EDTASM-PLUS. There are times when the user might like to determine the area of memory in which his program is to be assembled. The program, for example, might normally reside at 0C000H when it is loaded by the SYSTEM command in BASIC; it is convenient to be able to generate object code at this location so that the user can debug the actual locations for the program.

The O(rigin) command allows the user to specify this manual origin. Note that this is an EDTASM-PLUS **command** and not an assembler pseudo-operation. When the O command is entered, the Assembler responds with the following message:

```
*O
FIRST=7088  LAST=XFFE  USRORG=XFFF  USRORG=
```

The meaning of these parameters is shown below. FIRST is the first location after the Edit buffer, which is the first location of the symbol table for the program in the Edit buffer. LAST is the last location that the system can use. This is one less than the maximum RAM memory location. USRORG is the user specified manual origin. Initially this is the last location in memory.

A new USRORG may be specified by typing in a value between FIRST and LAST. When this is done, the user defines a **User Ram Area** into which he may assemble by the **manual origin switch, MO.** If an assembly into memory is then performed with a "/MO", the Assembler will assemble the object code starting at the user origin.

Suppose, for example, the user specifies a manual origin, or USRORG, of 8000H

```
*O
FIRST=7088  LAST=XFFE  USRORG=XFFF  USRORG=8000
```

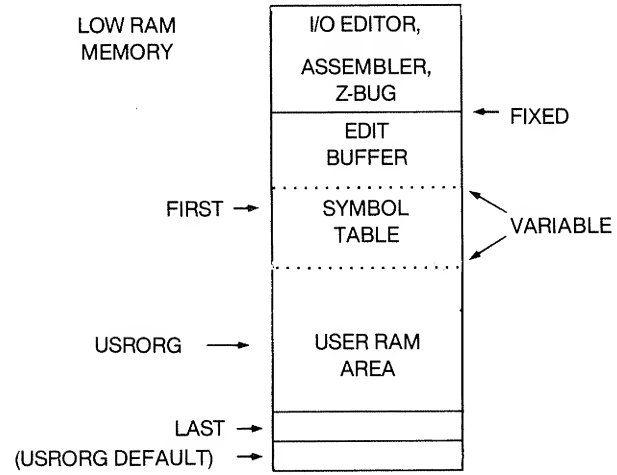
If he then starts an assembly by:

```
*A/IM/MO
```

the object code will be assembled into memory starting at location 8000H, and continuing upward from that location.

The **default value** for USRORG is the last available memory location. This value should be reentered if assemblies with automatic origins are to be done to provide maximum space for the symbol table and object code. For 16K RAM systems this value is 7FFF, for 32K systems BFFF, and for 48K systems FFFF.

As in automatic origin assemblies, no ORG is necessary. ORGs present in source code result in an origin value of the ORG operand plus the USRORG value, which is generally meaningless.



If the USRORG has been redefined by entering a new value after an "O" command, the area from USRORG to LAST becomes a **protected** memory area for **automatic** assemblies, similar to protecting RAM by the BASIC "MEMORY SIZE" option. If assemblies into memory are performed with automatic origin (no /MO switch), the assembler will only use the available RAM up to USRORG. The user RAM area may be used to hold other programs or data as desired,

The edit buffer is also limited by USRORG; it connot go beyond this location.

Assembling With an Absolute Origin

There is another switch associated with in memory assemblies, the **A**bsolute **O**rigin switch, /AO.

All assemblies that are performed that are **not** to memory (no /IM switch) must have an absolute origin by means of an ORG statement as we have seen. For example, to assemble a program that will load at 8000H, we specify

```
ORG    8000H        ;PROGRAM ORIGIN
```

and assemble by means of *A/LP, or other "A" command with switches.

The default condition for non-memory assemblies is an assembly with absolute origin and is equivalent to

*A/AO

All assemblies in which the object code is written to cassette tape are absolute assemblies.

EDTASM-PLUS is primarily designed for **in-memory** assemblies with the origin determined automatically by EDTASM-PLUS or the user (/MO). In these cases **no** user ORG pseudo-ops are required. It is possible to perform **in-memory absolute** assemblies with user-specified ORGs, but this technique is not recommended.

Absolute assemblies may be performed **in memory** by the sequence

*A/IM/AO or *A/IM/AO/MO

When an absolute origin assembly is done, ORG pseudo-ops after the first are **not** relative to the starting address of the program as they are when /AO is not used. In an absolute origin assembly, the source line

```
ORG    $+100        ;LEAVE TABLE AREA
```

will adjust the Assembler location counter by 100 as we described earlier in this chapter.

Assembler Expression Evaluation

EDTASM-PLUS includes many new **operators** and **expression evaluation** features. Expressions may be used in the assembly to simplify the programmer's task by automatically generating such things as table lengths, string lengths, constants, addresses, and other data. Let us see how these operators ...uh...operate.

Addition and subtraction are represented by "+" and "−", respectively. Some examples of their use are shown in the code below:

```
TABLE   DEFB    23+80H          ;FLAG AND CONSTANT
        DEFB    25+80H
        DEFB    17+80H
        DEFB    35+80H
TABSZ   DEFB    $-TABLE         ;FIND SIZE OF TABLE
```

In this example a table of constants is generated, with bit 7, the most significant bit, being set for each entry. Adding the 80H to set the bit is less abstract than calculating the value by hand for the DEFB operand.

The size of the table is nicely defined by the expression "$-TABLE", which generates a symbol TABSZ that is equal to the number of entries in TABLE. Additional entries may be added to the table (by reassembling) without having to refigure the size. TABSZ can be used as immediate data:

```
        LD      B,TABSZ         ;LOAD SIZE OF TABLE
```

Multiplication and division are represented by "*" and "/", respectively. Their use is less frequent than addition and subtraction, but in certain cases they may be very handy.

One example of the use of multiply might be the generation of a table size based on two variables, NENT, the number of entries in the table, and ENTSZ, the number of bytes per entry. These parameters could easily be changed by reassembling without affecting calculations related to the size of the table.

```
        LD      BC,NENT*ENTSZ   ;COMPUTE TABLE SIZE
```

The example below uses the division operator to find the starting point for a binary search based on table size.


```
        LD      HL,NENT*ENTSZ/2+TABLE
```

When a divide is performed, the division is done in integer fashion. If the result is not an integer, as in 13/2, the quotient is used as the result and the remainder is ignored. In general, expressions used in assemblies must be able to be equated to a 16-bit integer.

Logical operators allow the user to perform logical ANDs, ORs, and exclusive ORs on address and constant data in assemblies. These operators are used infrequently, but may be quite useful for generating certain types of data. The logical OR function is represented by ".OR." or "!", the logical AND by ".AND." or "&", and the logical exclusive OR by ".XOR." These operators must be used between two terms in an expression.

```
0000 0A    00100    DEFB  0A.OR.2     ;LOGICAL OR
0001 0B    00101    DEFB  0A!3        ;LOGICAL OR
0002 07    00110    DEFB  1FH.AND.7   ;LOGICAL AND
0003 07    00111    DEFB  1FH&7       ;LOGICAL AND
0004 18    00120    DEFB  1FH.XOR.7   ;LOGICAL EXCLUSIVE OR
0000       00130    END
00000 TOTAL ERRORS
```

The .NOT. operator takes the one's complement of a value or expression. The .NOT. is used before **one** term.

```
0000 AA    00100    DEFB  .NOT.55H    ;ONE'S COMPLEMENT
0001 55    00110    DEFB  .NOT.AAH    ;ONE'S COMPLEMENT
```

The shift operator is represented by a less than symbol (<). The shift operator may be used to generate data shifted **left** any number of bit positions by a **positive** shift count or **right** any number of bit positions by a **negative** shift count.

```
7173 38    00100    DEFB  7<3         ;SET BITS 5,4,3
7174 001E  00110    DEFW  3C00H<-1     ;GENERATE WORD
                                           ADDRESS
0000       00120    END
00000  TOTAL ERRORS
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

The modulo operator is represented by a ".MOD." The modulo operator is used to find the remainder result of a division of two numbers. The remainder result is then used as the data value.

```
0000 02    00100    DEFB    56.MOD.6    ;56/6 IS 9 REMAINDER 2
0001 03    00110    DEFB    111.MOD.9   ;111/9 IS 12 REMAINDER 3
0000       00120    END
00000 TOTAL ERRORS
```

The equals (.EQU.) and not equals (.NEQ.) operators are used between two terms in an expression. If the result of the expression is true, a 0FFH value is generated; if the result is false, a 0 value is produced. These operators find limited use, but may be used in the conditional assembly feature discussed next in this chapter.

```
0000 FF    00100    DEFB    1.NEQ.0
0001 00    00120    DEFB    1.NEQ.1
0000       00130    END
00000 TOTAL ERRORS
```

Parentheses "( )" may be used to group parts of an expression **if the first operator in an expression is not a left parenthesis.** For example,

```
LD HL,3*(5+2)
```

is valid, but

```
LD HL,(5+2)*3
```

is invalid and will cause an error.

Constants may be used in an expression, of course. Constants with an O or Q suffix are octal constants, constants with an H suffix are hexadecimal, and unsuffixed constants or those with a T suffix are decimal. An ASCII character is always bracketed by single quotes.

```
77H = 167O = 167Q = 119T = 119
'A' = ASCII A
```

The Assembler location counter is represented by a dollar sign ($) or period (.).

The operators that may be used in assembly, and their order of precedence are shown on the next page. The order of precedence is used to determine which parts of the expression will be evaluated first.

	Operator	Operation	Precedence Level
Lowest Precedence	.EQU. (or =)	equals	1
	.NEQ.	not equals	1
	+	addition	2
	−	subtraction	2
	.OR. (or !)	logical OR	3
	.XOR.	logical exclusive OR	3
	.AND. (or &)	logical AND	4
	*	multiplication	5
	/	division	5
	<	shift	5
	.MOD.	modulo	5
Highest Precedence	.NOT.	NOT (1's complement)	6
	( )	parentheses	7

Operations of the same precedence are applied left to right.

Conditional Assembly

EDTASM-PLUS includes two pseudo-ops that we have not discussed up to this point. The COND(itional) and ENDC(onditional) pseudo-ops permit **conditional assembly.** The formats of the COND and ENDC pseudo-ops are

```
COND    (expression)    (label not allowed)
ENDC                    (label not allowed)
```

If the value of the expression is **non-zero** (true), then the source code between the COND and the matching ENDC is assembled; if the value of the expression is zero (false), then the source code between the COND and ENDC is not assembled.

Let us see how this is utilized.

```
COND    M48K          ;ASSEMBLE IF 48K RAM
LD      SP,0C000H     ;INITIALIZE STACK POINTER
LD      HL,0AF80H     ;INITIALIZE BUFFER POINTER
ENDC                  ;END CONDITIONAL
```

In the example on the preceding page, the two load instructions are assembled if and only if symbol M48K is equal to a non-zero value. Typically this would be done by:

```
M48K    EQU     1               ;SET 48K
```

Any number of conditional assembly sets may be used; they may even be nested as in

```
        COND    M48K            ;ASSEMBLE IF 48K RAM
        LD      SP,0C000H       ;INITIALIZE STACK POINTER
        LD      HL,0AF80H       ;INITIALIZE BUFFER POINTER
        COND    CASS            ;ASSEMBLE IF CASSETTE
        LD      A,1             ;SET CASSETTE FLAG
        LD      (CASSF),A       ;STORE
        ENDC
        ENDC
```

Every COND must have a corresponding ENDC, and they may be nested up to 255 levels (although that is somewhat outrageous!).

Suppose that we have two sets of conflicting code. One set is to be assembled if memory is 64K while the other is not assembled for 64K, and vice versa. We can use the ".NEQ." operator to assemble either one set or the other:

```
        COND    M64K.NEQ.0      ;ASSEMBLE FOR 64K SYSTEMS
        LD      SP,0            ;INITIALIZE STACK POINTER
        ENDC

        COND    M64K.EQU.0      ;ASSEMBLE FOR NON-64K SYSTEMS
        LD      SP,7F00H        ;INITIALIZE STACK POINTER
        ENDC
```

If M64K is non-zero, the stack pointer is loaded with 0, while if M64K is zero, the stack pointer is loaded with 7F00H.

We now come to a philosophical question...Why do we want to conditionally assemble code? For some very good reasons. Suppose that you are in the software business producing TRS-80 software for various configurations of systems. Some systems have one cassette and one disk and 48K of RAM, other systems have four disks, no cassettes, and 32K of RAM, and so forth. It is very convenient to write one master version of a program that will handle all possible configurations of systems by conditionally assembling only that code which is applicable to the individual system.

The alternative to a master version is a separate version for each configuration. If this approach is used, **every** version must be changed if new code is put in, or existing code modified. It is much more convenient and accurate to change only one master version and use conditional assembly.

When the conditional assembly approach is used, the symbols for each COND are usually conveniently grouped at the beginning of the listing, so that they may be modified to reflect the desired configuration before assembly:

```
;CHANGE THESE PARAMETERS FOR SYSTEM GENERATION
NODISK  EQU     1               ;NUMBER OF DISKS
NOCAS   EQU     1               ;NUMBER OF CASSETTES
MEMSZ   EQU     48              ;MEMORY SIZE
LP      EQU     1               ;LINE PRINTER=1, NONE=0
SERIAL  EQU     0               ;RS-232=1, NONE=0
;END OF SYSTEM EQUATES
```

Chapter Four

Assembler Macro Capability

- What Is a Macro?
- Using Parameters in Macros
- When Can Macros Be Used?
- Rules for Macro Definitions
- Rules for Macro Reference
- Suppressing the Listing of a Macro
- Macro Labels Using #$YM
- Further Samples of Macro Usage
- Symbol Table Codes
- Assembler Error Messages

What Is a Macro?

A macro is a way of automatically generating assembly-language instructions. One way to think of macros is as "**in-line**" **subroutines**. Suppose, for example, that in an assembly-language program that we are writing we use the HL register pair to hold 16-bit values that need to be shifted left n number of bit positions. If we hold a shift count in the B register, then an easy set of instructions to do this would be

```
00400 LOOP    ADD     HL,HL    ;SHIFT HL LEFT, LOGICAL
00410         DJNZ    LOOP     ;CONTINUE FOR N TIMES
```

This code performs a left shift on the HL register by adding it to itself n times, where n is a count of 1–15 in the B register. (The DJNZ instruction decrements the count in the B register and jumps to LOOP if the result of the decrement is **not** zero.)

Now, one way to generalize this function is to make it a subroutine by adding a RET(urn) instruction and moving it to an area of the assembly-language program where it can be CALLed every time we want to perform a shift of HL. Before each call, the B register is loaded with a count of 1 through 15 that represents the number of times HL is to be shifted. Two CALLs and the subroutine itself are shown in the code below:

```
00100         LD      B,3      ;LOAD B FOR SHIFT OF HL 3
                                  BITS
00110         CALL    SHFTHL   ;SHIFT HL
              .
              .

              .
00550         LD      B,10     ;LOAD B FOR SHIFT OF HL 10
                                  BITS
00560         CALL    SHFTHL   ;SHIFT HL
              .
              .
              .
10000 SHFTHL  ADD     HL,HL    ;SHIFT HL LEFT, LOGICAL
10010         DJNZ    SHFTHL   ;CONTINUE FOR N TIMES
10020         RET              ;RETURN TO CALLING CODE
```


NOTE. Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

In the code above, we have used three dots to represent intervening code between the two CALLs and the actual subroutine.

A second way of generalizing the shift HL function is to make it a **macro**. First, we define the two steps for the shift as a **macro definition** and give this definition a name:

```
00018 SHFTHL  MACRO            ;DEFINE MACRO
00020         ADD     HL,HL    ;SHIFT HL, LOGICAL
00022         DJNZ    $-1      ;CONTINUE FOR N TIMES
00024         ENDM             ;END DEFINITION
```

The first pseudo-op, MACRO, essentially told the assembler that the following code was a macro named "SHFTHL". The second pseudo-op, ENDM, defined the end of the macro.

Note that this code does not generate any instructions at this point — it is simply a definition that the assembler notes.

We can reference the macro by referring to the macro name. Each time we use the macro name in the opcode field of an assembly-language line, the assembler decodes the name as a macro reference, finds the corresponding macro definition, and generates the instructions that have been defined for that macro. The complete source code for two references to the macro and the macro definition is:

```
00018 SHFTHL  MACRO            ;DEFINE MACRO
00020         ADD     HL,HL    ;SHIFT HL, LOGICAL
00022         DJNZ    $-1      ;CONTINUE FOR N TIMES
00024         ENDM             ;END DEFINITION
00100         LD      B,3      ;LOAD B FOR SHIFT OF HL 3 BITS
00102         SHFTHL           ;SHIFT HL MACRO
00550         LD      B,10     ;LOAD B FOR SHIFT OF HL 10 BITS
00560         SHFTHL           ;SHIFT HL MACRO
01000         END
```

The assembled version of this source code shows that the macro references are **expanded** into sets of the two instructions defined in the macro definition. The macro definition itself is simply listed without generating any code at that point.

```
                00018 SHFTHL MACRO          ;DEFINE MACRO
                00020        ADD    HL,HL   ;SHIFT HL, LOGICAL
                00022        DJNZ   $-1     ;CONTINUE FOR N
                                              TIMES
                00024        ENDM           ;END DEFINITION
0000 0603       00100        LD     B,3     ;LOAD B FOR SHIFT
                                              OF HL 3 BITS
                00102        SHFTHL         ;SHIFT HL MACRO
0002 29                      ADD    HL,HL   ;SHIFT HL, LOGICAL
0003 10FD                    DJNZ   $-1     ;CONTINUE FOR N
                                              TIMES
                             ENDM           ;END DEFINITION
0005 060A       00550        LD     B,10    ;LOAD B FOR SHIFT
                                              OF HL 10 BITS
                00560        SHFTHL         ;SHIFT HL MACRO
0007 29                      ADD    HL,HL   ;SHIFT HL, LOGICAL
0008 10FD                    DJNZ   $-1     ;CONTINUE FOR N
                                              TIMES
                             ENDM           ;END DEFINITION
0000            01000        END
00000 TOTAL ERRORS
```

NOTE. Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

Note that in the macro definition, we **avoided using labels** other than the macro name itself. The reason for this is that as the Assembler simply reproduces the macro definition every time it encounters a macro reference, it would reproduce the labels as well. More than one macro reference would produce **doubly defined labels**, which would result in an Assembler error diagnostic message. For a technique to use labels inside macros, see the section in this chapter labeled "Macro Labels Using #$YM".

Using Parameters in Macros

Using the macro on the preceeding page, we can **easily and automatically generate the code** for the shift function anytime we reference the macro by using the name in the op code field of an assembly-language source line.

This technique can be used for **any number of lines of code** that are defined by a macro definition and for **any number of individual macros**. Each macro definition, of course, must have a unique name to be referenced.

Generating a set of predefined instructions is only a portion of the power of macro usage, however. We can generate not only a set of specific instructions, but sets of instructions that use **different parameters** expressed in the macro reference. Let us see how this works.

In the SHFTHL macro above, we had to load the B register with a count prior to referencing the macro. By using the count as a **parameter**, we can automatically generate the proper load B instruction. The macro definition for this is:

```
00018 SHFTHL MACRO   #COUNT     ;DEFINE MACRO
00020        LD      B,#COUNT   ;LOAD COUNT
00022        ADD     HL,HL      ;SHIFT HL, LOGICAL
00024        DJNZ    $-1        ;CONTINUE FOR N TIMES
00026        ENDM
```

The macro definition line here has one parameter expressed as an arbitrary name. This name is used in the macro definition code as a general parameter. When the macro is referenced by invoking it with the name SHFTHL, the reference also includes a parameter which will take the place of #COUNT in the macro expansion. A sample reference is:

```
00550        SHFTHL  10         ;SHIFT HL 10 BITS
```

and the expansion for this reference is:

```
00550        SHFTHL  10         ;SHIFT HL 10 BITS
             LD      B,10       ;LOAD COUNT
             ADD     HL,HL      ;SHIFT HL, LOGICAL
             DJNZ    $-1        ;CONTINUE FOR N TIMES
             ENDM
```

We have left out the source code generated by the macro expansion, but the reader can see that the LD B instruction has used the parameter of 10 from the macro reference line as the parameter to be used in the load. **The Assembler has automatically generated a set of instructions with the specified parameter.**

The number of parameters used in a macro may be any number that may fit into a source line. Let us try another example that illustrates how several parameters may be used. In this example, we will use the block move instruction of the Z-80, the LDIR. The LDIR moves a block of data from a **source** location to a **destination** location. Prior to the LDIR, the HL register is loaded with the source address, the DE register is loaded with the destination address, and the BC register is loaded with the number of bytes to move. A typical sequence is:

```
01000          LD      HL,17000   ;SOURCE ADDRESS
01010          LD      DE,18000   ;DESTINATION ADDRESS
01020          LD      BC,64H     ;100 BYTES IN BLOCK
01030          LDIR               ;MOVE SRCE BLK TO DEST BLK
```

The above code moves the bytes in locations 17000 through 17099 to locations 18000 through 18099.

Now, let us take the above code and make a general macro out of it, using the source address, destination address, and number of bytes to move as parameters. When we finish we will have a general macro that can be used anytime we want to move a block of data.

```
00050 MOVE     MACRO   #SRCE,#DEST,#NUM
00052          LD      HL,#SRCE   ;LOAD SOURCE ADDRESS
00054          LD      DE,#DEST   ;LOAD DESTINATION ADDRESS
00056          LD      BC,#NUM    ;LOAD BYTE COUNT
00058          LDIR               ;MOVE BLOCK
00060          ENDM
```

The above macro definition uses three parameters, #SRCE, #DEST, and #NUM, which are referenced inside the macro. To reference the MOVE macro, the reference line would include three parameters to be used in the macro expansion. These parameters could be any values, symbols, or expressions that would normally be used in coding the four instructions. Typical references for the MOVE macro would be

```
02000          MOVE    17000,18000,64H
```

which duplicates the "typical" sequence for LDIR we saw previously,


```
03000          MOVE    BUFF1,SCREEN,1024
```

which uses symbolic addresses for the source and destination, and

```
04000          MOVE    BUFF2+1024,3C00H,1024
```

which uses an expression for one of the parameters.

The macro expansion for the first example would be

```
02000          MOVE    17000,18000,64H
               LD      HL,17000   ;LOAD SOURCE ADDRESS
               LD      DE,18000   ;LOAD DESTINATION ADDRESS
               LD      BC,64H     ;LOAD BYTE COUNT
               LDIR               ;MOVE BLOCK
               ENDM
```

Parameter usage in the macro may involve simple loads as we have seen above, or may be much more complicated. There is no reason that we could not subtract two parameters, such as #ARG2–#ARG1, or perform other manipulations with the parameters, **just as we do in writing in-line code.**

When Can Macros Be Used?

What are the relative merits of macros versus subroutines? In the subroutine case the code is generated only one time and exists in one area of memory. Aside from the overhead of the CALL and RET(urn) instructions, subroutine usage is very efficient in terms of speed and extremely efficient in use of memory. When one must perform a set of predefined instructions over and over, subroutines may be used to advantage.

However, when parameters are involved, referencing the macro and having the Assembler automatically generate the proper code is extremely efficient in terms of program development time. Nobody likes to sit down and write long sequences of instructions, and EDTASM-PLUS makes the task much easier by the use of macros.

The one disadvantage of macros, of course, is the amount of memory that is used when many macros are expanded. In the early days of computers (five years ago), this was a critical factor. However, memory is very inexpensive today, and programming **time** is much more important. For all intents and purposes, the amount of memory used by macros can be ignored.

As a matter of fact, using macros in EDTASM-PLUS will probably result in **less** memory being used! This is because the edit buffer also uses memory space. One-time definitions of macros with macro references are much more efficient in terms of number of characters of source code than repeating the source code without using macros!

Macro usage has many exciting possibilities on the TRS-80. Because each macro definition may have many arguments, and there may be any reasonable number of macro definitions, whole applications **languages** may be defined and utilized. For example, it is not hard to visualize a high-speed word processing language written almost entirely in macros. We will whet your appetite by showing a sample of how this **could** be performed:


```
01000 ;THIS SEQUENCE DELETES A PARAGRAPH
01010 SCAN    PI,BACK   ;SCAN FOR PARAGRAPH MARK
01020 JP      M,NFND    ;GO IF NOT FOUND
01030 SCAN    PI,FWRD   ;SCAN FOR PARAGRAPH MARK
01040 JP      M,NFND    ;GO IF NOT FOUND
01050 DELS    CS,CE     ;DELETE SCREEN CHARACTERS
01060 DELB    BS,BE     ;DELETE BUFFER CHARACTERS
```

Another example of macro use is the **simulation** of another assembly-language on the TRS-80. Using macros, it is relatively easy to construct a **cross-assembler** to, say, assemble source code for the 6800 microprocessor on the TRS-80, or to assemble source code and to execute code for a purely hypothetical machine!

An example of macros for the 6800 is shown below:

```
02000 QE200   LDX    BUFFER   ;LOAD INDEX WITH ADDRESS
02010         LDAA   256      ;NUMBER OF BYTES
02020         ADDB   23       ;ADJUST B REGISTER
02030         BNE    QE302    ;GO IF NOT EQUAL
02040         JMP    QE125    ;GO FOR NEXT SET OF VALUES
```

The corresponding macro definition for the JMP macro would assemble the proper machine language code **for the 6800** by:

```
00100 JMP     MACRO   #ARG1          ;JMP DEFINITION
00101         DEFB    7EH            ;OP CODE FOR JMP
00102         DEFB    #ARG1/256      ;EXTENDED ADDRESS
00103         DEFB    #ARG1.MOD.256
00104         ENDM
```

Before showing some further examples of macro usage, let us first set down the rules for macro definitions and references.

Rules for Macro Definitions

As we have seen in the previous example, all macros in EDTASM-PLUS consist of a **macro definition line, a body of code, and an end macro line:**

```
NAME    MACRO     #P1,#P2,#P3,...#PN
        (body of macro)
        ENDM
```

The label, or macro name, is required and can be any valid assembly-language label. The parameters are optional depending upon the code in the body of the macro. Here are the rules for the parameters:

The first character of the dummy parameter must be a "#".

The maximum length of the dummy parameter string is 16 characters, including the "#".

Commas must be used between parameters.

The last parameter string must be terminated by a carriage return, space, tab, or semicolon.

Parameter strings may contain any characters except delimiters (",", space, tab, ";") or the "#" symbol.

The ENDM is always the last statement in the macro. It cannot have a label, or the Assembler will not recognize it as an end macro pseudo-op.

The body of the macro can consist of any valid assembly-language lines with Z-80 operation codes, pseudo-ops, operands, and comments. Normally, labels are not permitted in the body as they will cause doubly-defined labels if the macro is referenced more than once. (We will explain a special technique for label use later in this chapter.) Dummy arguments referenced within the body of the macro use the same strings as defined in the MACRO line.

Nested macros are not permitted. This is an important point. We **cannot** have a macro definition such as:


```
00030 SHIFT   MACRO   #ARG1     ;SHIFT MACRO
00031         LD      B,#ARG1   ;LOAD SHIFT COUNT
00032         ADD     HL,HL     ;SHIFT ONE BIT POS'N
00033 TEST    MACRO             ;TEST BIT 2
00034         BIT     2,C       ;SET FLAG
00035         ENDM
00036         DJNZ    $-3       ;LOOP IF NOT DONE
00037         ENDM
```

It is also illegal for a macro definition to contain a macro reference.

As we have seen, the macro definition does not generate any code at the time of definition. Code is only generated when the macro is referenced. The macro definition is normally listed with line numbers, but without generated code. This listing can be suppressed by the *LIST ON and *LIST OFF Assembler commands, just as any source lines may be controlled.

Two new Assembler commands, *MLIST ON and *MLIST OFF, allow the listing of macro expansions to be suppressed. We will discuss these in a later section.

Rules for Macro Reference

We have already seen a few examples of macro definition and expansion. One point that was implied, but may not be obvious, is that the **macro definition must come before any reference to it.** It is considered good assembly-language programming practice to group all of the macro definitions close to the beginning of the assembly-language program so that the Assembler will have processed the definitions before any reference to them. A "MACRO FWD REFERENCE" error occurs if this is not done.

The label on a macro reference line is optional and causes no problem if used, as it is not a part of the actual macro expansion.

Parameters are optional and depend upon the parameters used in the macro definition. The parameters are substituted in the same order as they are defined in the macro definition. The first parameter corresponds to the first dummy, the second to the second dummy, and so forth.

The rules for the parameters in the macro reference line are:

Each parameter string must be less than 16 characters.

Each parameter must be separated by commas.

The last parameter must be terminated by a space, carriage return, tab, or semi-colon.

Delimiters (space, semi-colon, tab, comma) may be used in a parameter string if they are enclosed by single quotes.

The last point may be somewhat confusing. If a text parameter is used in the body of the macro, then a delimiter, such as a blank imbedded in the text, may confuse the assembler.

An example of this is shown in the macro and reference on the opposite page which uses a DEFM, DEFine Message, within the body of the macro to assemble a text parameter. We would like to have the message "AN EVEN BREAK" assemble into the macro reference. The imbedded blank, however, fools the Assembler into terminating the parameter after "AN" and only the "A" and "N" are generated.


```
                00100 TEXT      MACRO   #ARG1
                00110           DEFM    '#ARG1'
                00120           ENDM
                00130           TEXT    AN EVEN BREAK
0000    41                      DEFM    'AN'
0001    4E
                                ENDM
0000            00140           END
00000 TOTAL ERRORS

TEXT    652C
```

This error can be remedied by enclosing the entire parameter in quotes in the macro reference. The assembler then treats the text as a single parameter and the message is assembled properly.

When a single quote must be generated itself, a special case arises. The single quote is denoted by two successive single quotes. The text string "ABC'DEF" must be coded in the macro reference line above as 'ABC''DEF' with quotes around the entire text and a **pair** of single quotes to represent the desired single quote.

Suppressing the Listing of a Macro

It is often desirable to **suppress the listing** of macro expansions. This is not done so much to keep proprietary code from prying eyes (although this **is** a use), but to reduce the listing to a more manageable and readable form. The *MLIST OFF command sets the no print mode for macro expansions and the *MLIST ON restores printing. The MLIST commands are completely separate from the LIST commands. The *LIST commands control all assembly line printing while the *MLIST commands control only printing for macro expansions. In the example below note that the "LD D,4" instruction is not in a macro expansion and is printed even though the macro expansion is not.

```
                    00100 PRINT   MACRO   #ARG1,#ARG2,#ARG3
                    00110         LD      A,#ARG1
                    00120         LD      B,#ARG2
                    00130         LD      C,#ARG3
                    00140         ENDM
                    00150 *MLIST OFF
0000    1604        00160         LD      D,4
0002                00170         PRINT   1,2,3
                    00180 *MLIST ON
0008                00190         PRINT   4,5,6
0008    3E04                      LD      A,4
000A    0605                      LD      B,5
000C    0E06                      LD      C,6
                                  ENDM
0000                00200         END
00000 TOTAL ERRORS
```

Macro Labels Using #$YM

Every time a macro is expanded, an **implicit** parameter named #$YM is generated. In effect, this parameter is set equal to the number of times all macros have been expanded. If five macros have been used since the start of assembly, #$YM = 5; if 15 macros have been expanded since the start of assembly, #$YM = 15.

As the value of #$YM is unique for each macro expansion, it may be used to create a label for a macro without causing doubly defined labels. Let us see how this works.

Every time #$YM is used in a macro, a four-character string representing the current value (macro count) of #$YM is substituted in place of the string #$YM. Labels can therefore be created in the macro by defining such strings as A#$YM or B#$YM. When the macro is expanded, these labels will become A000, A0003, B0015, or other unique labels for the macro.

The example below shows the use of #$YM in creating labels for two macros, each of which is expanded twice.

```
7000                00100         ORG     7000H
                    00110 FIRST   MACRO
                    00115 A#$YM   JP      A#$YM
                    00120         ENDM
                    00130 SECND   MACRO
                    00135 B#$YM   JP      B#$YM
                    00140         ENDM
                    00150         FIRST                 ;#$YM = 0000
7000    C30070              A0000 JP      A0000
                                  ENDM
                    00160         SECND                 ;#$YM = 0001
7003    C30370              B0001 JP      B0001
                                  ENDM
                    00170         FIRST                 ;#$YM = 0002
7006    C30670              A0002 JP      A0002
                                  ENDM
                    00180         SECND                 ;#$YM = 0003
7009    C30970              B0003 JP      B0003
                                  ENDM
0000                00190         END
00000 TOTAL ERRORS
```

Further Samples of Macro Usage

In this section we will illustrate further use of macros by two examples, a macro for subroutine calls, and a macro to fill a portion of the screen with a given character.

Using macros for subroutine calls combines the advantages of both macros and subroutines. Subroutines take up minimal memory, but may require a great deal of overhead in loading parameters before a CALL is actually made to the subroutine.

In this example, we define a macro that sets up the necessary parameters for a CALL to a subroutine that performs disk reads and writes. The parameters that must be initialized before the subroutine is called are:

Disk drive number in A register

Sector number of read or write in B register

Track number of read or write in C register

Function in D register: 0 = read, 1 = write

Buffer address in HL register

Normally these must be defined by performing five loads before the CALL to DISKIO, but this macro permits us to write the parameters in a single line:

```
                00100 DISKIO   MACRO    #ARG1,#ARG2,#ARG3,
                                          #ARG4,#ARG5
                00110          LD       A,#ARG1
                00120          LD       B,#ARG2
                00130          LD       C,#ARG3
                00140          LD       D,#ARG4
                00150          LD       HL,#ARG5
                00160          CALL     DISKDR
                00170          ENDM
                00180 ; EXAMPLE OF EQUATE FOR FUNCTIONS
0000            00190 READ     EQU      0
0001            00200 WRITE    EQU      1
                00210 ; EXAMPLE OF MACRO USE FOR READING
                        TRACK 10, SECTOR 5 INTO
```



```
                00220 ; BUFFER AT 8000H FROM DISK DRIVE 1
                00230 ;                          (Single line call)
                00240          DISKIO   1,5,10, READ,8000H
7235  3E01                     LD       A,1
7237  0605                     LD       B,5
7239  0E0A                     LD       C,10
723B  1600                     LD       D,READ
723D  210080                   LD       HL,8000H
7240  CD4372                   CALL     DISKDR
                               ENDM
                00250 ; DUMMY ADDRESS FOR DISKDR
                        SUBROUTINE
7243            00260  DISKDR  EQU      $
0000            00270          END
00000 TOTAL ERRORS
```

(Macro expansion)

NOTE. Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

The macro for filling a screen segment uses four arguments for the screen starting and ending position and one argument for the fill character. When the macro is expanded, the HL register pair is loaded with the screen address of the starting line and character position (#ARG1 * 64 + #ARG2 + 3C00H), the BC register pair with the screen address of the ending line and character position (ARG#3 * 64 + ARG4 + 3C00H), and the A register with the character to be filled. This character is filled to the screen as long as HL and BC are not equal; HL is incremented by one each time through the loop.

Two #$YM type labels are used within the macro to prevent duplicate references for expansion of more than one macro.

```
                00100 FILLSC   MACRO    #ARG1,#ARG2,#ARG3,
                                          #ARG4,#ARG5
                00110          LD       HL,#ARG1 * 64 + #ARG2
                                          + 3C00H
                00120          LD       BC,#ARG3 * 64 + #ARG4
                                          + 3C00H
```

```
                    00130           LD      A,'#ARG5'
                    00140 D#$YM     LD      (HL),A
                    00150           PUSH    HL
                    00160           OR      A
                    00170           SBC     HL,BC
                    00180           POP     HL
                    00190           JR      Z,F#$YM
                    00200           INC     HL
                    00210           JR      D#$YM
                    00220 F#$YM     EQU     $
                    00230           ENDM
                    00240 ; SAMPLE CALL TO FILL LINE 7,
                            CHARACTER POSITION 7
                    00250 ; THROUGH LINE 9, CHARACTER POSITION
                            23 WITH A CHARS.
                    00260 ;         (single line invokes macro)
                    00270           FILLSC  7,7,9,23,A
721E  21C73D                        LD      HL,7*64+7+3C00H
7221  01573E                        LD      BC,9*64+23+3C00H
7224  3E41                          LD      A,'A'
7226  77              D0000   LD      (HL),A
7227  E5                      PUSH    HL
7228  B7                      OR      A
7229  ED42                    SBC     HL,BC
722B  E1                      POP     HL
722C  2803                    JR      Z,F0000
722E  23                      INC     HL
722F  18F5                    JR      D0000
7231            F0000   EQU     $
                        ENDM
0000            00280   END
00000 TOTAL ERRORS
```

(macro expansion)

NOTE. Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

Symbol Table Codes

Every time an assembly is run, the **symbol table** is compiled by the Assembler and listed (if the /NS switch is inactive). The Assembler uses special codes to help the user recognize what various labels in the symbol table represent.

An undefined code ("U") is listed if the symbol is erroneously referenced but never defined.

```
UNDEFINED SYMBOL
0000 3E00       00100           LD      A,VALUE   ;LOAD VALUE
0000            00110           END
00001 TOTAL ERRORS

VALUE 0000 U
```

A macro code ("M") is listed if the symbol is a macro name.

```
                00100 EXMPLE    MACRO
                00110           LD      A, 20
                00120           ENDM
                00130           EXMPLE
0000 3E14                       LD      A, 20
                                ENDM
0000            00140           END
00000 TOTAL ERRORS

EXMPLE          M
```

A forward reference code ("F") is listed if the symbol is a macro name referenced **before** definition of the macro.

```
BAD OPCODE
00090           EXMPLE
MACRO FWD REF
00100 EXMPLE    MACRO
                00110           LD      A, 20
                00120           ENDM
0000            00140           END
00002 TOTAL ERRORS

EXMPLE          UMF
```

The redefined symbol code ("R") is listed if the symbol is erroneously defined more than one time.

```
MULTIPLE DEFINITION
0000 3E14        00100 LOOP   LD      A, 20
MULTIPLE DEFINITION
0002 3E14        00110 LOOP   LD      A,20
0000             00120        END
00002 TOTAL ERRORS

LOOP  0000       R
```

The DEFLed symbol code ("D") is listed if the symbol is defined by a DEFL pseudo-op.

```
000A             00100 EXMPLE DEFL    10
0014             00110 EXMPLE DEFL    20
0000             00120        END
00000 TOTAL ERRORS

EXMPLE  0014     D
```


Assembler Error Messages

The following messages are used in the assembly listing to inform the user of **assembly errors**. The error message is used just prior to the source line in which the error occurs. The source line is then displayed or printed following the error message.

Message	Description and Corrective Action
BAD ADDRESS	Invalid address defined for USRORG command or address above LAST or below USRORG on assembly. Use an address between LAST and FIRST.
BAD ADDRESSING MODE	Operands use incorrect addressing mode. Use valid addressing for the instruction.
BAD EXPRESSION	Syntax of expression in source line incorrect. Redefine.
BAD MEMORY	Assemble into memory verify does not compare. Check RAM.
BAD LABEL	Invalid label has been used. Use 1 to 6 character label, starting with an alphabetic character. Do not use "reserved" (system) words; check label.
BAD OPCODE	Invalid opcode or pseudo-op mnemonic has been used. Check spelling.
BRANCH OUT OF RANGE	Relative address displacement on JR or other instruction is greater than +127 or less than −128. Use JP or another instruction.
DIVISION BY 0	Expression used 0 as a divisor. Use non-zero value.
ENDC WITHOUT COND	ENDC encountered without a prior COND. Check for existence of COND.
ENDM WITHOUT MACRO	ENDM encountered without a prior MACRO. Check for existence of MACRO.
FIELD OVERFLOW	Instruction field cannot hold value used as operand. Change value to fit.
MACRO FWD REF	Macro invoked before definition. Define macro before reference.

Message	Description and Corrective Action
MISSING INFORMATION	Operands missing or incomplete in source line. Check format of instruction or pseudo-op.
MULTIPLY DEFINED SYMBOL	Reference to multiply defined symbol (see next error).
MULTIPLE DEFINITION	Same label was used again. Change to one unique label for each line.
NESTED MACROS	Nested macros are not allowed. Redefine macros.
NO END STATEMENT	No END pseudo-op at end of source code. Add END.
STACK OVERFLOW	Expression too involved for stack. (Should rarely occur.)
SYMBOL TABLE OVERFLOW	Too many labels used in source program. Shorten or delete labels (use $).
SYNTAX ERROR	Macro syntax incorrect. Check.
UNDEFINED SYMBOL	Symbol encountered that is not defined as label, EQU, DEFL, or MACRO. Define.

Chapter Five

EDTASM-PLUS Z-BUG

- **Basic Z-BUG Operation**
- **Displaying and Modifying Memory Locations in Hexadecimal**
- **Displaying and Modifying Memory Location in Decimal and Octal**
- **Symbolic Debugging**
- **Free Use of Symbols, Expressions, and Constants**
- **Displaying a Block of Locations**
- **Displaying and Modifying Registers and Flags**
- **One Time Type Outs and Examining the Addressed Location**
- **Breakpointing**
- **Single Stepping Through a Program**
- **Loading and Saving System Format Tapes**
- **Loading Stand-Alone Z-BUG**
- **Z-BUG Cautions and Error Messages**

Basic Z-BUG Operation

The Z-BUG portion of EDTASM is a newly designed, powerful debug package that can be used to debug in-memory assemblies or any other machine language code.

Some of the functions that Z-BUG can perform are:

A display of the contents of memory locations in byte form, word form, ASCII form, and Z-80 mnemonic form.

Modification of memory locations by many of the above formats

Breakpointing of up to eight locations

Single-stepping through programs

Symbolic referencing of symbols in the assembly symbol table

Calculator mode operations

Because Z-BUG is normally resident together with the Assembler and Editor, the entire EDTASM-PLUS package is extremely interactive. After errors are found by using Z-BUG, the source program in the Edit buffer may be modified, a rapid in-memory assembly performed, and another level of debug carried out, all without time-consuming cassette activity.

In the following chapter we will explain the capabilities of Z-BUG in detail and use an example of actual program development.

Z-BUG is entered by entering "Z" while in the system command mode. After receiving control, Z-BUG prints the prompt character "#" as shown below:

```
*Z
Z-BUG
#
```

To return to the system command mode at any time, type an "$E" after the # prompt (the "$" is a [SHIFT] [↑]).

```
#$E
*
```


Displaying and Modifying Memory Locations in Hexadecimal

When Z-BUG is initially entered, the "examination mode" is **mnemonic**. Any locations that display in this mode are in the form of **Z-80 mnemonics**. The following sequence shows the effects of this mode:

```
Z-BUG
#0/      DI        (enter [↓] )
 1/      XOR A     (enter [↓] )
 2/      JP 674    (enter [↓] )
 5/      JP 4000   (enter [ENTER] )
#
```

As you can see from the sequence, a location is examined by typing in the value of the location, followed by a slash [/] . Subsequent locations are examined by entering a down arrow [↓] .

In this mode, consecutive **instructions** are displayed, rather than consecutive bytes. If an instruction is three bytes long, for example, the Z-BUG location pointer is incremented to the next **instruction** for display.

The mnemonic examination mode can be set at any time by entering an "$M" after a Z-BUG "#" prompt. **The dollar sign character is not a dollar sign, but is an ESCAPE character, represented by a** [SHIFT] [↑] . Many Z-BUG instructions use the ESCAPE in this fashion.

```
#$M          (set mnemonic examination mode)
  ↑
  └── [SHIFT] [↑]  prints as $
```

There are three other examination modes in Z-BUG. If $B is entered at any time, the examination mode is set to **byte mode**. Byte mode displays the data in memory one byte at a time with a value of 0 through 0FFH displayed as the byte contents. As in the preceding example, we can look at consecutive bytes by using the down arrow [↓] .

```
#$B
#0/     0F3
 1/     0AF
 2/     0C3
 3/     74
 4/     6
```

Word mode is set by entering an $W after a # prompt. Word mode displays two consecutive bytes as a **16-bit word**. This mode is used to look at locations containing memory addresses or 16-bit data. The display of data is a value from 0 through 0FFFFH, with the data arranged second byte, first byte in standard Z-80 address orientation. Compare these locations with the display from byte mode, for example:

```
#$W
#0/     0AFF3
 2/     74C3
```

If the specified address in word mode starts on an odd location, odd numbered words are displayed:

```
#$W
 1/     0C3AF
 3/     674
```

ASCII mode is set by entering a $A after a # prompt. ASCII mode displays the memory locations one byte at a time in ASCII. (The examination mode is set to byte and the type-out mode to ASCII.)

```
#$A                          (Byte Mode Equivalent)
#111/   R                    #111/   52
 112/   A                     112/   41
 113/   D                     113/   44
 114/   I                     114/   49
 115/   O                     115/   4F
 116/                         116/   20
 117/   S                     117/   53
 118/   H                     118/   48
```

In the example above, a valid ASCII message is displayed, as shown. Z-BUG will attempt to display the 7-bit ASCII equivalent, even if the contents of the memory locations do not represent an ASCII message, as in:


```
#$A
#0/     S                    0/     0F3
 1/     /                    1/     0AF
 2/     C                    2/     0C3
```

To display any memory location using Z-BUG, set the proper examination mode by typing in $M, $B, $W, or $A. This mode will remain in force until it is again changed. Then enter any hexadecimal address followed by a slash [/].

We have a number of options during the display of memory locations. If we want to keep on displaying consecutive locations we can do so by entering a down arrow [↓], as we have seen. If we want to display the **previous** location we can enter an up arrow [↑], as shown in this sequence:

```
#100A/      71      ( [↑] )
 1009/      41      ( [↑] )
 1008/       1      ( [↑] )
 1007/      20
```

Regardless of the examination mode, **the up arrow [↑] displays the last byte.** This means that use of the up arrow in the mnemonic and word modes may be somewhat ambiguous, unless the up arrow is simply used for positioning the current location back to a new starting point.

If we want to stop the display of consecutive locations at any time we can "close" the current location by typing in an [ENTER], bringing us back to the # prompt of Z-BUG:

```
#1007/      20          ( hit [ENTER] )
#
```

To modify the current memory location type in the new value in hexadecimal, followed by a down arrow [↓] or [ENTER]. If the examination mode is the byte mode, mnemonic mode, or ASCII mode, one or two hexadecimal digits can be entered, with any leading hex "A" through "F" prefixed by a zero:

```
#8001/  5       12      (changes 5 to 12H)      [↓]
 8002/  7       0C      (changes 7 to 0CH)      [↓]
 8003/  0AA     0BB     (changes 0AA to 0BBH)   [ENTER]
#
```

If the examination mode is the word mode, up to four hexadecimal digits can be entered, with an appropriate leading zero for non-numeric digits:

```
#8000/  0FFFF   1234   (changes FFFF to 1234H)   ( ↓ )
 8002/  0FFFF   5678   (changes FFFF to 5678H)   ( ↓ )
 8004/  0FFFF   9ABC   (changes FFFF to 9ABCH)   ( ↓ )
 8006/  0FFFF   0DEFA  (changes FFFF to 0DEFAH)  ENTER
#
```

To modify the current location to an ASCII character while in **any mode,** type in the ASCII character with a leading and trailing single quotation mark:

```
#8000/   —   'N'
 8001/   —   'O'
 8002/   —   'W'
 8003/   —   ' '
 8004/   —   'I'
 8005/   —   'S'
```

The slash [/] may be used by itself without a location value preceding it to reopen the "current" location for any mode.

Displaying and Modifying Memory Locations in Decimal and Octal

EDTASM-PLUS has the capability to not only display or accept input in hexadecimal, but to handle data in decimal or **octal.** Octal is base 8 format which is not frequently used on the Z-80, and we will confine most of our discussion here to decimal display and input.

There are two control words that determine the number base used in type outs and inputs. The command "$O" sets the **O**utput radix (number base) control word to 8, 10, or 16. The command "$I" sets the **I**nput radix to any radix between 2 and 16.

The format of the $O command is

(radix)$O

where radix is 8, 10, or 16. The default radix is 16, and this is the output format when Z-BUG is initially entered. The radix may be changed at any time by entering the $O command.

To see how this works, look at the following sequence, which first prints several locations in hexadecimal, and then prints the same locations in decimal and octal.

```
#16$O
#1000/    20      ( enter ↓ )
 1000/    28      ( enter ↓ )
 1002/     7      ( ENTER )
#10$O
#1000/    32T     ( enter ↓ )
 4097T/   40T     ( enter ↓ )
 4098T/    7T     ( ENTER )
#8$O
 1000/    40Q     ( enter ↓ )
 10001Q/  50Q     ( enter ↓ )
 10002Q/   7Q     ( ENTER )
#
```

The suffix of "T" denotes a decimal number as we can see above, while a "Q" is used to represent an octal number.

It is important to note that the **input** radix remains hexadecimal while the output radix is changed. The input and output radices are completely separate and set by two different commands. Of course, in most cases the two will be the same radix to avoid confusion.

The format of the $I command is

(radix)$I

where radix is any number from 2 through 16. The default radix is 16, and this is the format when Z-BUG is initially entered. The radix may be changed at any time by the $I command.

The following sequence sets both the input and output radix to decimal and then modifies the decimal locations to the values shown. Note that the "T" suffix is used after every **display** of a decimal number.

```
#10$I
#10$O
#32000T/    255T    123     (change 255 to123) ( ↓ )
 32001T/    255T    156     (change 255 to 156) ( ↓ )
 32002T/    255T    189     (change 255 to 189) ( ↓ )
 32003T/    250T    111     (change 250 to 111) ( ENTER )
#
```

Although the input radix can be any number from 2 through 16, most input operations we would typically want to perform would be either decimal, for table data, or hexadecimal, for instruction data. Occasionally it might be useful to enter data in binary when bit processing is involved. Entering data in base 7 or 13 would probably have somewhat more limited applications.

The suffixes of "T" for decimal, "H" for hexadecimal, and "O" or "Q" for octal may be used anytime data is entered, regardless of the input radix setting. If we are in the hexadecimal input mode (16$I), for example, we can enter hexadecimal data by typing hexadecimal digits (with a leading zero for non-numeric), decimal data by typing decimal digits with a "T" suffix, or octal data by typing octal digits with a "Q" suffix:

```
#16$O
#16$I
#8000/      12      0AA     (change to 0AAH) ( ↓ )
 8001/      0FF     12      (change to 12H)  ( ↓ )
 8002/      15      12T     (change to 12
                            decimal)         ( ↓ )
 8003/      0A5     99T     (change to 99
                            decimal)         ( ↓ )
 8004/      32      77Q     (change to 77 oc-
                            tal)             ( ENTER )
```

The slash [/] may be used at any time to reopen the "current" location.

Symbolic Debugging

EDTASM-PLUS permits another way of entering or displaying data. If an in-memory assembly has been run and Z-BUG is entered, then we may choose to make **symbolic references.** In this mode of operation, we can refer to locations in the in-memory object code by their **labels** instead of by reference to absolute locations.

This is a very powerful feature of EDTASM-PLUS, made possible by the fact that the symbol table is resident in memory at the same time as the object code and debugger.

Let us investigate how this type of reference functions. We will use the same program we used in an earlier chapter on in-memory assembly.

```
71AD 21003C     00110 CLEAR   LD    HL,3C00H  ;ADDRESS OF
                                                  SCREEN START
71B0 010004     00120         LD    BC,1024   ;COUNT OF POSI-
                                                  TIONS
71B3 3E20       00130 LOOP    LD    A,20H     ;LOAD BLANK
71B5 77         00140         LD    (HL),A    ;STORE BLANK
71B6 23         00150         INC   HL        ;BUMP POINTER
71B7 0B         00160         DEC   BC        ;DECREMENT
                                                  COUNT
71B8 78         00170         LD    A,B       ;TEST DONE
71B9 B1         00180         OR    C         ;MERGE LS BYTE
71BA 20F7       00190         JR    NZ,LOOP   ;GO IF MORE TO
                                                  STORE
0000            00200         END
00000           TOTAL ERRORS
CLEAR           71AD
LOOP            71B3
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a second line with the information continued beneath the last column.

This program has two labels, "CLEAR" and "LOOP". The instructions corresponding to the two labels are at 71ADH and 71B3H, respectively (these addresses may vary dependent upon the version of EDTASM-PLUS in use).

The symbolic debugging mode is set by the Z-BUG command "$S". It is also the default (initial) mode. To get back to the **numeric** mode, a second command, "$N", is used. When the numeric mode is in force, all data will be displayed as in the previous examples.

If we have assembled the program above in memory (A/IM), then we can set the symbolic mode and reference locations by their symbolic names:

```
#$S
#$B
 #CLEAR/        21          ( ↓ )
  CLEAR +1/     0           ( ↓ )
  CLEAR +2/     3C          ( ↓ )
  CLEAR +3/     1           ( ↓ )
  CLEAR +4/     0           ( ↓ )
  CLEAR +5/     4           ( ↓ )
  LOOP/         3E          ( ↓ )
  LOOP+1/       20          ( ↓ )
  LOOP+2/       77          ( ↓ )
  LOOP+3/       23          ( ENTER )
 #
```

Initially we typed in the "CLEAR"; for every following location Z-BUG supplied the proper symbolic name of the location.

Note that Z-BUG references every location to the last symbol if no new label is found. When a new label is found, consecutive locations are referenced to the new label.

Of course, we can use the symbolic mode with any **examination** mode; they are two separate functions:

```
#$S
#$M
#CLEAR/         LD HL,3C00          ( ↓ )
CLEAR+3/                LD BC,400   ( ↓ )
LOOP/           LD A,20
```

We can also enter **expressions** that include symbolic terms at any time. For example if we were in the **word examination mode** we could **alter** location LOOP + 10 (out of the program) to a 16-bit address of LOOP + 4!


```
#$W
#$S
#LOOP+10/       0       LOOP+4      ( ENTER )
#LOOP+10/       LOOP+4              ( ENTER )
#$N
#71B3+10/       71B7
LOOP
```

In the sequence above, we first set word mode ($W) and symbolic mode ($S). We then examined location LOOP + 10H and modified the location with the value of LOOP + 4. When we again examined LOOP + 10 it had the value of LOOP + 4 in it. In numeric format ($N) LOOP is equal to 71B3, and LOOP + 4 is 71B7.

In processing the above, Z-BUG looked up the symbol LOOP in the symbol table to try to resolve every input reference using it. In the symbolic mode, Z-BUG also prints out every output in terms of values in the symbol table, if it finds a symbol in the approximate range of the value involved. If no symbol is in the approximate range, Z-BUG reverts back to output of values in whatever numeric radix is in effect.

Any operator normally used for assemblies may be used in an expression, except that ".DIV." must be used for divide as "/" is used to open a location for display.

The use of "$" to indicate the current location (as in the Assembler) is **not recommended** as it may be confused with an ESCAPE. Use a period (".") instead.

Z-BUG has an **expression evaluator** command, "=", that allows a calculator-style typeout of all types of expressions. Normally it can be used to find the locations of a symbol plus or minus a displacement, but the command may be used for any expression, including those for hexadecimal arithmetic. Some examples are shown here:

```
#5+0EF=0F4
#TABLE+200=71B5
#8014-15E=7EB6
#4AA+10T+377Q=5B3
```

Symbolic debugging can be used with great convenience in examining and modifying tables of data, variables, and constants, and in using some of the other Z-BUG commands we will be discussing shortly.

Free Use of Symbols, Expressions and Constants

Z-BUG is extremely flexible in allowing use of symbols at any time, regardless of input or output mode. A non-numeric string that is input to open a location or to modify a location will cause Z-BUG to scan the current symbol table (if any) to find an equivalent value for the string.

Likewise, any **expression** may be used to open or modify locations. The expressions may contain symbols, constants, and operators in any combination.

Suffixes may be used after numeric data for opening or modifying a location regardless of the input radix. Suffixes may also be used in expressions in any combination. Of course, no suffix is necessary for inputting data in the current input radix.

Displaying a Block of Locations

A block of locations may easily be displayed in Z-BUG by using the $T command. The format of $T is

(first address)␣(last address)$T where ␣ is a blank

This command will **T**ype out the block from first address through last address in whatever examination mode and output radix has been specified.

```
#1000 100E$T
 1000/    20
 1001/    28
 1002/    7
 1003/    78
 1004/    0B9
 1005/    0E
 1006/    2A
 1007/    20
 1008/    '
 1009/    41
 100A/    71
 100B/    0D7
 100C/    28
 100D/    14
 100E/    0FE
#
```

The [SHIFT] [@] or [BREAK] may be used to hold or abort output for the T command.

Displaying and Modifying Registers and Flags

The $R command of Z-BUG causes a **display of all registers,** except the R register, in hexadecimal. (The R register changes its value with each new instruction for memory refresh and examining its contents is somewhat meaningless.)

The format of the register display is

```
#$R
 A=XX  F=XX=      B=XX C=XX D=XX E=XX H=XX L=XX
 A'=XX F'=XX=     B'=XX C'=XX D'=XX E'=XX H'=XX L'=XX
 SP=XXXX  PC=XXXX   IX=XXXX   IY=XXXX   I=XX
#
```

Individual registers or register pairs can also be examined and modified by entering the register or register pair name followed by the "open" symbol, " / ".

```
#A/                    255T
#HL/                   30000T
```

The register or register pair names follow standard Z-80 mnemonics — A, B, C, D, H, L, BC, DE, HL, SP, PC, IX, IY, or I, or their primed equivalents.

It is not recommended that the PC or SP registers be modified by the user. The PC is used to hold the address of the current instruction of the user's program for G(o) and single-step commands and changing the PC during debugging may be disastrous. The SP points to the user's stack area and changing the SP may be equally bad.

Registers can be examined after a **breakpoint** in an object program to see that they hold expected results. They may be modified before continuing from a breakpoint or **single step** or before transferring control back to an object program routine. We will show some examples of this action later in the chapter.

Flags may be examined by typing F/ or by using the $R command. The format of the flags printout is

```
=SZHP/VNC
```

If the individual flags are **set** they will be listed; if the individual flags

are **reset** they will not be listed. The display

```
= S P / V
```

for example, indicates that the S and P/V flags = 1 and that Z, H, N, and C = 0.

One Time Type Outs and Examining the Addressed Location

There are four additional Z-BUG commands that are used during examination of memory, the ";" command, the " = " command, the right cursor → command, and the ":" command.

The ";" command is used after the Z-BUG prompt or after displaying a location with a slash. It forces the contents of the current location to be displayed in numeric mode and would typically be used to temporarily display the numeric value of a location while in the symbolic mode.

If we were in mnemonic mode ($M), for example, we would be displaying instruction mnemonics. If we also were in symbolic mode ($S), we would be displaying the instructions with symbolic addresses rather than absolute addresses. In the case of our previous program example, a display of the instruction at 71BAH would result in:

```
#$M
#$S
#71BA/          JR NZ,LOOP
```

If we wanted to find out the equivalent absolute address for "LOOP" we could immediately enter a ";" after the mnemonic typeout to obtain

```
#71BA/          JR NZ,LOOP ; JR 71B3
```

The " = " command is used in similar fashion, either after a "#" to display the current location or after display caused by a slash. It displays the current location in byte examination mode ($B) and numeric mode ($N) regardless of the current examination and numeric/symbolic mode.

The " = " command would typically be used when we were confused about the contents of a location displayed in mnemonic or symbolic form. This could easily happen if we were in mnemonic examination mode and we started displaying data from a table, for example. The data might well be converted to equivalent instructions, and we could print out the data in numeric form by entering the " = " command.

```
#8000/          RST 38      =0FF
```

The ↵ command is used to open a new location based on the addressed location in the current instruction (mnemonic mode) or the address represented by the current location and the current location plus one.

It is convenient to be able to perform this action for a number of reasons. In the case of instructions that reference memory locations, the ↵ can be used to examine the contents of the locations that are being loaded or used for storage or to follow the sequence of instructions for instructions that jump to other locations. When in byte, word, or ASCII mode, the ↵ may be used to examine the location pointed to by the address found in the two current bytes; this is handy for examining routines or data pointed to by tables of addresses.

Suppose that we are in mnemonic mode and are examining an instruction that loads a variable "COUNT". We can examine the contents of COUNT by entering a right arrow after the mnemonic print out:

```
#START/         LD A,(COUNT)        ( ↵ )
COUNT/          LD HL,0FFFF   =21
```

In the sequence above, START was examined in mnemonic mode. The instruction at START referenced COUNT, and by typing ↵ location COUNT was opened and printed out in mnemonic form. We then converted the mnemonic form into numeric form by " = ", which printed the first byte, or the contents of COUNT. As another example, suppose that we had been in word examination mode and investigating a table of subroutine addresses. We could open the location corresponding to any of the addresses by using ↵ :

```
#TABLE/         70FC            ( ↵ )
 70FC/          203E
#$M
#/
 70FC/          LD A,20
```


The " : " command is used to display the current location in flags mode format (= SZHP/VNC). This command would normally be used to examine a byte representing Z-80 flags, such as stack data created by PUSH AF.

Breakpointing

A **breakpoint** is very similar to a STOP command in BASIC; it is a way of breaking out of program execution at a predetermined point in the program.

The breakpoint is invaluable during program debugging because:

1. It informs us that we **reached** the breakpoint
2. It allows us to examine intermediate results in registers or memory.

Let us see how breakpointing can be used with Z-BUG. We will use the program we have been working with in this chapter and the chapter on in memory assembly:

```
71AD 21003C     00100 CLEAR  LD   HL,3C00H  ;ADDRESS OF
                                               SCREEN START
71B0 010004     00110        LD   BC,1024    ;COUNT OF POSI-
                                               TIONS
71B3 3E20       00120 LOOP   LD   A,20H      ;LOAD BLANK
71B5 77         00130        LD   (HL),A     ;STORE BLANK
71B6 23         00140        INC  HL         ;BUMP POINTER
71B7 0B         00150        DEC  BC         ;DECREMENT
                                               COUNT
71B8 78         00160        LD   A,B        ;TEST DONE
71B9 B1         00170        OR   C          ;MERGE LS BYTE
71BA 20F7       00180        JR   NZ,LOOP    ;GO IF MORE TO
                                               STORE
0000            00190        END
00000 TOTAL ERRORS
CLEAR 71AD
LOOP  71B3
```

NOTE: Some lines which fit as one line on your computer screen will not fit on this manual page as one line. These lines have been continued on a

second line with the information continued beneath the last column.

We would like to execute the program to see if it actually **does** clear the screen. We will put in a **breakpoint** at LOOP so that we can check the registers (we have never learned to completely trust computers. . .).

Z-BUG allows up to eight separate breakpoints, numbered 0 through 7. The format of the command to **set a breakpoint** is:

(address)$X

The address value can be a symbolic address, a numeric address, or a symbolic expression. We could, for example, use

#LOOP$X or #71B3$X or #CLEAR+6$X

Z-BUG uses the **next available breakpoint number**. Since no other breakpoints are in force at this point, Z-BUG uses breakpoint number 0. We can **display all breakpoints** used at any time by the command $D:

```
#LOOP$X
#$D
 0 @ LOOP
```

The display above tells us that one breakpoint is in effect @ LOOP, and that the number of the breakpoint is 0.

Now we are ready to **execute** the program. We can execute from any address by using the $G command of Z-BUG. The format of the $G command is

(address)$G

Here again the address may be symbolic, numeric, or an expression. We will execute starting at CLEAR:

```
#CLEAR$G
 0 BRK @ LOOP
#
```

The $G command caused program execution starting at CLEAR. The two instructions at CLEAR and CLEAR+3 were executed, and a breakpoint then occurred at LOOP **before execution of the instruction at LOOP.**

At this point we can perform any operations that we think might be useful, including examining registers or memory, changing registers or memory, or setting additional breakpoints. We will display the registers to allay some of our fears about computers:

```
#$W
#HL/     3C00
#BC/     400
#
```

After making any examinations that we desire, we can then C(ontinue) from the current breakpoint by the $C command. The format of the $C command is:

(continue count)$C where (continue count) is interpreted as decimal

The continue count tells Z-BUG not to stop at the current breakpoint until the current breakpoint has been reached the number of times equal to the count. If no count is used, a default value of 1 is assumed. If we continued by $C, we would **execute the instruction at LOOP**, and then continue execution of the program. The program in this case brings us back to the breakpoint at LOOP

```
#$C
 0 BRK @LOOP
#
```

We could stop at the 500th time through the loop by 500$C as shown here:

```
#500$C
 0 BRK @LOOP
#BC/          20B
#
```

The count in BC has been decremented down to 523 at this point (notice that the upper portion of the screen has been cleared).

Before we continue, we had better put in a breakpoint at the end of the program to guarantee our return to Z-BUG. If this is not done, we will never regain control. The last instruction of the program is at 71BAH. We should like to breakpoint at the last instruction plus one instruction. Although **there is no instruction there** (there is **garbage** of some type), we can breakpoint at 71BC anyway and simulate

the non-existent instruction.

```
#71BC$X
#$D
 0 @ LOOP
 1 @ LOOP+9
#
```

At this point there is probably no need for the breakpoint at LOOP. We can delete it by another Z-BUG breakpoint instruction, $Y(ank). The $Y can be used to yank all breakpoints by entering:

```
#$Y
```

We can also selectively yank or delete breakpoints by specifying a breakpoint number for deletion:

```
#0$Y
```

The command above deletes the breakpoint at LOOP, number 0. When a breakpoint is yanked, it is released to the pool of eight possible breakpoints and could immediately be used again. The number specified before the $Y is always interpreted as a decimal number.

We now have one breakpoint, at LOOP+9:

```
#$D
 1 @ LOOP+9
#
```

If we continue from this point, we will go through the remainder of the loop, clearing the screen and finally "fall through" to the breakpoint at LOOP+9. At that point we can delete the breakpoint and continue further debugging, if necessary.

```
#$C
 1 BRK @ LOOP+9
#$Y
#
```

Up to 8 breakpoints can be used in the fashion above to generally cover all paths through the program for debugging. The combination of multiple breakpoints and "continue for n times" is a powerful debugging tool that should let the user have complete control over program checkout.

Single Stepping Through a Program

Z-BUG also provides the capability to **single-step** through a program. In this mode, one instruction at a time is executed with a display of the next instruction to be executed and the contents of that location.

The format of the single step command is

```
(address)@
```

To initially start the single step sequence, a numeric or symbolic address can be specified as in:

```
#CLEAR@
 CLEAR+3/ LD BC,400
#
```

Thereafter only the "@" is necessary to continue stepping through sequential instructions:

```
#CLEAR@
 CLEAR+3/ LD BC,400
#@
 LOOP/ LD A,20
#@
 LOOP+2/ LD (HL),A
```

At any time after the display of the current instruction, other Z-BUG commands may be used, such as commands to examine registers and memory locations. Single stepping may thus be used to **trace** the flow of the program and the contents of registers and memory locations.

Single stepping may be used after a breakpoint. ESCAPE "C" without a continue count can be used to resume program execution after single stepping.

Loading and Saving System Format Tapes

Z-BUG has the capability of saving any RAM memory area on cassette tape under a file name in "SYSTEM" format. A tape file created in the $P(unch) command may be reloaded by either the Z-BUG $L(oad) command or by the SYSTEM command in Level II BASIC.

The $P command can be used to save "patched" (partially debugged) assembly-language programs, several programs that have been merged into one file, data areas, or any other convenient set of instructions or data or both.

The format of the $P command is

(first)␣(last)␣(execution)␣NAME$P where ␣ is blank

First is the first memory location to be saved, last is the last memory location to be saved, execution is the starting address (if any), and NAME is a 1 to 6 character file name. If NAME is omitted, the name "NONAME" is used.

After the $P string has been entered, Z-BUG displays the message "READY CASSETTE". When the cassette is positioned, any key can be pressed and the block of locations from first to last is written to cassette.

If no execution address is required, simply type in a "dummy" address for the above sequence.

To reload the saved file under Z-BUG enter

```
#NAME$L
```

If no file name is used, Z-BUG will reload the **next** cassette file into memory. After the load, the Z-BUG prompt character "#" will be displayed.

The starting address on the tape is loaded into the user PC register (examine it by "PC/").

If a checksum error occurs during the load, a "C" will be displayed in place of the "tape load" asterisk; if a memory verify fails, an "M" will be displayed. In either failure case, the tape will **continue** the load until the end of the cassette file.

Z-BUG will skip over files that do not have the correct file name. Previous problems in certain TRS-80 software relating to tape searches for named files are not present in EDTASM-PLUS, and more than one named file may be stored on cassette (and also retrieved).

Z-BUG uses the same format for cassette files as Level II BASIC SYSTEM mode, so files P(unched) under Z-BUG can be loaded under Level II BASIC by:

```
>SYSTEM          (enter SYSTEM mode)
*?NAME           (read cassette file)
*?/              (transfer to "execution" address)
```

EDTASM-PLUS object files created by the EDTASM-PLUS Assembler may also be loaded by the Z-BUG L command.

Loading Stand-Alone Z-BUG

Z-BUG may be loaded alone, without the Editor and Assembler, to provide more room for large assembly-language programs that are to be debugged.

To load Z-BUG in this "stand-alone" fashion, position the EDTASM-PLUS cassette tape directly before the Z-BUG file (SIDE TWO) and load using the SYSTEM command:

```
>SYSTEM          (enter SYSTEM mode)
*?ZBUG           (load Z-BUG)
*?/              (start execution)
```

The MICROSOFT COPYRIGHT notice, followed by "STAND-ALONE Z-BUG" will be displayed, followed by the Z-BUG prompt character "#". Normal Z-BUG (not Editor or Assembler!) operations can now be carried out.

Stand-Alone Z-BUG occupies about 7K bytes of RAM with the user RAM area starting at about 5B80H. Do not attempt to use RAM memory below 5B80H.

Z-BUG Cautions and Error Messages

Z-BUG utilizes **ReSTart** (RST) instructions as either calls to ROM routines or breakpoint control, and therefore **RSTs cannot be used in user programs** that are run under Z-BUG.

When a user executes a program under Z-BUG, an **internal** (to Z-BUG) stack is utilized as a **user** stack. This stack area is approximately 50 bytes long and will suffice for most user programs. If the user's program contains many levels of stack use, however, he should set up his own stack area in the user RAM by performing an

```
LD          SP,TOPSTK
```

where "TOPSTK" is a user-defined "top-of-stack" address. This top of stack will usually be the maximum memory location available **plus one**, but may be any high memory area away from the object code.

Z-BUG error messages include "BAD EXPRESSION", "BAD MEMORY", "DIVISION BY 0", "STACK OVERFLOW", and "UNDEFINED SYMBOL" with the same meanings as the corresponding Assembler messages.

The "ZERR" message is also used to denote one of the following conditions:

1. Expression followed by semicolon instead of an equal sign.
2. Illegal ESCAPE command character.
3. Internal breakpoint problem.
4. Illegal input or output radix specification.
5. Attempting to set more than eight breakpoints.
6. Attempting to set a breakpoint at a location where one already exists.
7. Attempting to breakpoint a register or register pair.
8. Attempting to breakpoint an RST instruction.
9. Specifying a continue count ($C) of 0.
10. Attempting to continue ($C) without being in a breakpointed condition.

General Index

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Class Permit No. 108 Bellevue, WA

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

Attn: Dept. D

REGISTRATION CARD—Please register your product with us so that we may keep you informed of new Microsoft products and program development.

Name _____

Address _____

City _____ State _____ Zip _____

Age _____ Occupation _____

Computer type _____

Peripheral equipment _____

Which Microsoft program did you purchase? _____

Comments _____

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

Catalog No. 1104
Part No. 10F04

Printed in U.S.A.